Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Logične strukture in sistemi

Drugi del

Andrej Dobnikar

Ljubljana, marec 2009

CIP - Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

004.312:621.382.3(075.8)

DOBNIKAR, Andrej
Logične strukture in sistemi / Andrej Dobnikar ; [izdajatelj] Fakulteta za računalništvo in informatiko. - 1. izd. - Ljubljana : Založba FE in FRI, 2009

ISBN 978-961-6209-73-1 (zv. 2, Fakulteta za računalništvo in informatiko)

244409856

Recenzenta: dr. Uroš Lotrič, dr. Branko Šter
Založnik: Založba FE in FRI, Ljubljana
Izdajatelj: Fakuleta za računalništvo in informatiko, Ljubljana
Urednik: mag. Peter Šega

Natisnil: Fakulteta za računalništvo in informatiko, Ljubljana
Naklada: 50 izvodov
1. izdaja

Kazalo

Poglavje 1

UVOD

Po teoretičnih osnovah v prvem delu o preklopni logiki, tranzistorski tehnologiji, funkcijah in osnovnih postopkih njihove sinteze, je v drugem delu govora o organizaciji in arhitekturi računalnika, o podrobnem opisu posameznih modulov ter o njihovem snovanju.

Na začetku je seveda govora o načinu računanja, ki ima osnovo v dvojiškem številskem sistemu. Najprej je predstavljeno pretvarjanje med številskimi sistemi, sledi predstavitev negativnih števil (tri variante) in pretvarjanje z omejeno natančnostjo. Na tej osnovi je mogoče podati strukturne značilnosti binarnih operacij, n.pr. seštevanja, množenja in deljenja. Pomembno vlogo ima tudi zapis v plavajoči vejici, ki predstavlja kompromis med natančnostjo in obsegom zapisa realnih števil, kot jih hrani računalnik. Pri realnih implementacijah funkcijskih modulov število vhodov velikokrat presega dovoljeno število, kar pomeni, da je potrebno izvajati sintezo večnivojske logike, s katero rešujemo problem vhodov. Tedaj pa nastopijo težave s hazardi, ki jih je potrebno eliminirati s posebnimi pravili snovanja. Tudi sekvenčna vezja v praksi predstavljajo določene težave, predvsem v zvezi s sinhronizacijo, ki jih rešujemo s protokoli. Opisani so najpomembnejši sekvenčni moduli, od splošnih in pomikalnih registrov, števcev do pomnilnikov. Podana je razlika med Mealyjevimi in Moorovimi stroji, opisana minimizacija stanj, kodiranje stanj in delitev strojev na manjše gradnike. Sledi opis organizacije računalnika, predvsem pomnilniškega vmesnika, vhodno/izhodnega vmesnika in strategije povezav, od točkovne, preko enojnega vodila do večvodilnih povezav. V zadnjem poglavju pa je podana implementacija računalniškega krmilnika na več

načinov, z naključno logiko, s pomočjo dekompozicije stroja, s skočnim števcem, skočnim sekvenčnikom in na osnovi mikroprogramiranja.

Učbenika Logične strukture in sistemi, prvi in drugi del, predstavljata celoto. Prvi del je bolj teoretičen in podaja temelje logike in snovanja, drugi pa uporablja pridobljeno znanje v prvem delu za razlago delovanja in ustroja računalnika ter dodaja nekatera znanja, ki so potrebna pri praktičnemu snovanju računalniških gradnikov, tako kombinacijskih kot sekvenčnih.

Poglavje 2

ŠTEVILSKI SISTEMI

Poznamo mešane in nemešane sisteme števil. Mešani imajo različno množico cifer na posameznem utežnem mestu, nemešani pa enako.

Mešani sistem:

$0 \div 7$	$0 \div 1$	$0 \div 3$	množice cifer
2	1	0	indeks

$$N = a_2u_2 + a_1u_1 + a_0u_0$$

$$u_0 = 1$$

$$u_1 = 4$$

$$u_2 = 4 \cdot 2 = 8$$

Nemešani sistem:
Številski sistem z osnovo R:

$$N = r_mR^m + r_{m-1}R^{m-1} + \ldots + r_1R^1 + r_0R^0 + r_{-1}R^{-1} + r_{-2}R^{-2} + \ldots + r_{-k}R^{-k}$$

$$N = r_mr_{m-1}\ldots r_1r_0, r_{-1}r{-}2\ldots r_{-k}$$

2.1 Pretvarjanje številskih sistemov

iz N v N'

$$N = a_ma_{m-1}\ldots a_1a_0, a_{-1}a_{-2}\ldots a_{-k}\ldots (\text{osnova } b)$$

$$N' = A_m A_{m-1} \dots A_1 A_0, A_{-1} A_{-2} \dots (\text{osnova } B,\ B \neq b)$$

2.1.1 Celi del

Deljenje se izvaja v sistemu b.

$$x_1 = \left[\frac{N}{B}\right] \rightarrow A_0 = N - x_1 B$$

$$x_2 = \left[\frac{x_1}{B}\right] \rightarrow A_1 = x_1 - x_2 B$$

$$\vdots$$

$$x_{i+1} = \left[\frac{x_i}{B}\right] \rightarrow A_i = x_i - x_{i+1} B$$

$$\vdots$$

$$x_{m+1} = \left[\frac{x_m}{B}\right] \rightarrow A_m = x_m - x_{m+1} B$$

[.] predstavlja zaokrožitev navzdol

2.1.2 Decimalni del

Množenje se izvaja v sistemu b.

$$A_{-1} = [B \cdot N] \rightarrow x_1 = B \cdot N - A_{-1}$$

$$A_{-2} = [B \cdot x_1] \rightarrow x_2 = B \cdot x_1 - A_{-2}$$

$$\vdots$$

$$A_{-i} = [B \cdot x_{i-1}] \rightarrow x_i = B \cdot x_{i-1} - A_{-i}$$

$$\vdots$$

PRIMER 1: $152_{10} = X_3$

3	152			
	50	2	=	A_0
	16	2	=	A_1
	5	1	=	A_2
	1	2	=	A_3
	0	1	=	A_4

$$152_{10} = 12122_3$$

PRIMER 2: $152_{10} = X_{13}$

13	152			
	11	9	=	A_0
	0	11	=	A_1

$$152_{10} = (11)9_{13}$$

PRIMER 3: $0,625_{10} = X_2$

				$0,625$
				2
A_{-1}	=	1	,	25
A_{-2}	=	0	,	50
A_{-3}	=	1	,	00

$$0,625_{10} = 0,101_2$$

PRIMER 4: $0,4_{10} = X_2$

				$0,4$
				2
A_{-1}	=	0	,	8
A_{-2}	=	1	,	6
A_{-3}	=	1	,	2
A_{-4}	=	0	,	4
			$\vdots$	*se ponavlja*

$$0,4_{10} = 0,\ \underbrace{0110}_{\text{se ponavlja}}\ \underbrace{0110}_{\text{se ponavlja}}\ \ldots\ _2$$

Pri ponavljanju decimalnega dela se postavlja vprašanje, koliko cifer v novem številskem sistemu potrebujemo za dosego določene natančnosti.

PRIMER 5: $0,4_{10} = X_2$, $napaka < 0,01_{10}$

$$2^{-k} < 0,01$$

$$-k \cdot \log_2 2 = \log_2 0,01$$

$$-k = \log_2 0,01$$

$$k = -\log_2 0,01 = -\frac{\log_{10} 0,01}{\log_{10} 2}$$

$$\lceil k \rceil = 7$$

Pretvorba logaritma: $\log_a p = \frac{\log_b p}{\log_b a}$

Pri pretvarjanju števil z osnovo različno od 10 v osnovo različno od 10 postopimo: ($b \neq 10 \neq B$; $b \neq B$)

$$N_b \rightarrow N_{10}$$

$$N_{10} \rightarrow N_B$$

PRIMER 6: $20,2_3 = X_5$ z natančnostjo $0,01_{10}$

- $20,2_3 = 2 \cdot 3^1 + 0 \cdot 3^0 + 2 \cdot 3^{-1} = 6,66\ldots_{10}$
- $6_{10} = X_5$

5	6			
	1	1	=	A_0
	0	1	=	A_1

  $6_{10} = 11_5$

- $0,66_{10} = X_5$

				0,66
				5
A_{-1}	=	3	,	30
A_{-2}	=	1	,	50
A_{-3}	=	2	,	50
A_{-4}	=	2	,	50
				$\vdots$

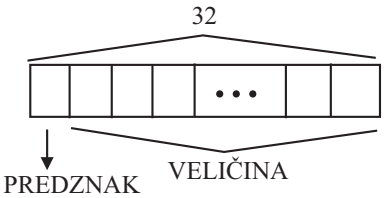
  $0,66_{10} = 0,3122\cdots_5$

- $5^{-k} = 0,01 \rightarrow -k \cdot \log_5 5 = \log_5 0,01$
  $k = -\log_5 0,01$
  $k = -\frac{\log_{10} 0,01}{\log_{10} 5} = 2,8614$
  $\lceil k \rceil = 3$ (navzgor zaokroženo)

- $20,2_3 = 11,312_5$ natančnostjo $0,01_{10}$

2.2 Predstavitve negativnih števil

2.2.1 Predznak - veličinska predstavitev

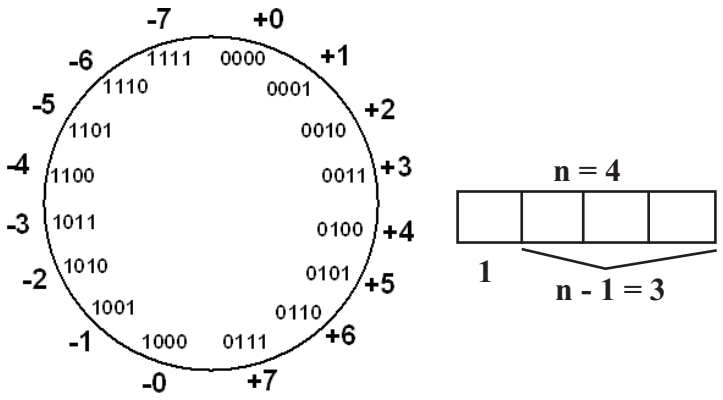


MSB (Most significant bit) predstavlja predznak: 0 pomeni pozitivno število, 1 pa pomeni negativno število. Ostali biti predstavljajo veličino.

Danes so števila običajno predstavljena z 32 biti.

Na sliki 2.1 je številsko kolo za predznak-veličinsko predstavitev za $n = 4$, skupaj z decimalnimi ekvivalenti.

- Največje pozitivno število: $+7 = 2^3 - 1$
- Najmanjše negativno število: -7
- Dve predstavitvi 0: +0, -0
- Seštevanje $N_1 + N_2$:
  - enaka predznaka: veličini seštejemo in dodamo predznak. Če se predznak spremeni, je OVERFLOW (prekoračitev, preliv).



Slika 2.1: Številsko kolo za predznak veličinsko predstavitev

PRIMER 1:

$$(+4) + (+3):$$

$$\begin{array}{rc|cccl} & 0 & 1 & 0 & 0 & \\ + & 0 & 0 & 1 & 1 & \\ \hline & 0 & 1 & 1 & 1 & = +7 \end{array}$$

$$(-4) + (-3):$$

$$\begin{array}{rc|cccl} & 1 & 1 & 0 & 0 & \\ + & 1 & 0 & 1 & 1 & \\ \hline & 1 & 1 & 1 & 1 & = -7 \end{array}$$

$$(+5) + (+3):$$

$$\begin{array}{rc|cccl} & 0 & 1 & 0 & 1 & \\ + & 0 & 0 & 1 & 1 & \\ \hline & 1 & 0 & 0 & 0 & \text{OVERFLOW} \end{array}$$

Carry povzroči OVERFLOW

- različna predznaka: manjšo veličino odštejemo od večje in dodamo predznak večje veličine. Tukaj se uporablja operacija odštevanja!

PRIMER 2:

$$(+4) + (-3):$$

$$\begin{array}{rc|cccl} & 0 & 1 & 0 & 0 & \\ - & 1 & 0 & 1 & 1 & \\ \hline & 0 & 0 & 0 & 1 & = +1 \end{array}$$

$$(-4) + (+3):$$

$$\begin{array}{rc|cccl} & 1 & 1 & 0 & 0 & \\ - & 0 & 0 & 1 & 1 & \\ \hline & 1 & 0 & 0 & 1 & = -1 \end{array}$$

Pri PREDZNAK-VELIČINSKEM sistemu potrebujemo tako seštevalnik kot odštevalnik, 0 je predstavljena z dvema kombinacijama: +0 in -0.

PRIMER 3:

$$n = 6$$

$$(+19) + (-10)$$

$$\begin{array}{rc|ccccl} & 0 & 1 & 0 & 0 & 1 & 1 & \\ - & 1 & 0 & 1 & 0 & 1 & 0 & \\ \hline & 0 & 0 & 1 & 0 & 0 & 1 & = +9 \end{array}$$

$$n = 6$$

$$(-11) + (+9)$$

$$\begin{array}{rc|ccccl} & 1 & 0 & 1 & 0 & 1 & 1 & \\ - & 0 & 0 & 1 & 0 & 0 & 1 & \\ \hline & 1 & 0 & 0 & 0 & 1 & 0 & = -2 \end{array}$$

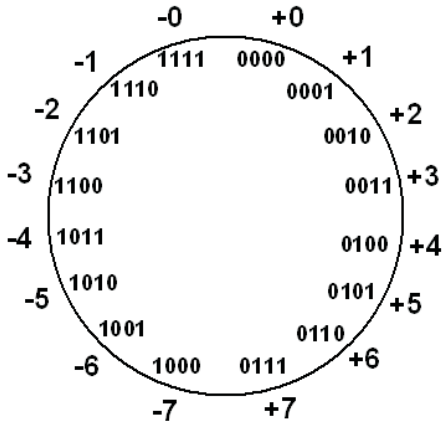
2.2.2 Eniški komplement (1'komplement)

Na sliki 2.2 vidimo primer številskega kolesa za 1'komplement za $n = 4$: Pozitivna števila imajo enako predstavitev kot pri PREDZNAK-VELIČINSKEM sistemu.

Negativno število $\overline{N}$ dobimo iz pozitivnega N, če negiramo vse bite: $\overline{N} = (2^n - 1) - N$.

$n = 4$:

$$N = 0100 \quad (+4)$$



Slika 2.2: Številsko kolo za 1'komplement

$$\overline{N} = 1111 - 0100 = 1011 \quad (-4)$$

• Negativna števila imajo 1 v prvem bitu (kot pri PREDZNAK-VELIČINSKI predstavitvi)

• Še vedno sta 2 predstavitvi za 0 ($+0 = 0000$ in $-0 = 1111$)

• Odštevanje se prevede na kombinacijo seštevanja in negacije: $A - B = A + (-B)$, negacija pa je enostavna (zamenjava 0 in 1) - torej ne rabimo odštevalnika

• Dve predstavitvi 0 pa zakomplicirata seštevanje → zato vpeljemo 2' komplement.

• Seštevanje N_1 in N_2:

- oba predznaka + : enako kot pri PREDZNAK-VELIČINSKEM sistemu

  PRIMER 1: $(+4) + (+3)$ (Če se predznak spremeni → OVERFLOW)

$$\begin{array}{rcccc} & 0 & 1 & 0 & 0 \\ + & 0 & 0 & 1 & 1 \\ \hline & 0 & 1 & 1 & 1 \quad = +7 \end{array}$$

- oba predznaka - : (EAC (End Around Carry) se vedno pojavi in ga dodamo k vsoti)

PRIMER 2: $(-4) + (-3)$

		1	0	1	1	
	+	1	1	0	0	
EAC →	(1)	0	1	1	1	↘
		0	1	1	1	vsota
	+				1	(EAC)
		1	0	0	0	= - 7

- različna predznaka, pozitivno število absolutno večje od negativnega: (EAC dodamo k vsoti, enako kot prej)

  PRIMER 3: $(+4) + (-3)$

		0	1	0	0	
	+	1	1	0	0	
EAC →	(1)	0	0	0	0	↘
		0	0	0	0	vsota
	+				1	(EAC)
		0	0	0	1	= + 1

- različna predznaka, negativno število absolutno večje od pozitivnega: (nerodno seštevanje)

  PRIMER 4: $(-4) + (+3)$

	1	0	1	1	
+	0	0	1	1	
	1	1	1	0	= -1

Pri različnih predznakih N_1 in N_2 se EAC pojavi le, če je pozitivno število večje v absolutnem smislu od negativnega.

• Zakaj korekcija z EAC deluje?

Vzemimo: $\underline{M + (-N)}$, $M > N$

$$M - N = M + \overline{N} = M + (2^n - 1 - N) = (M - N) + 2^n - 1$$

Z operacijo EAC pa odštejemo 2^n in prištejemo 1.

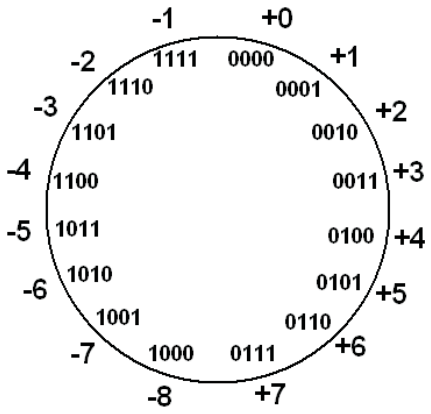
Enako velja pri: $\underline{(-M)+(-N)}$, $M+N<2^{n-1}$

$(-M)+(-N)=\overline{M}+\overline{N}=(2^n-M-1)+(2^n-N-1)=2^n+\underbrace{[2^n-1-(M+N)]}_{-(M+N)}-1$

Zopet sledi, da z EAC odštejemo 2^n in prištejemo 1, kar daje $-(M+N)$

2.2.3 Dvojiški komplement (2'komplement)

Na sliki 2.3 vidimo številsko kolo za 2'komplement pri $n=4$:



Slika 2.3: Številsko kolo za 2'komplement ($n=4$).

Kot vidimo, imamo tukaj samo eno ničlo(+0)! Zato pa pridobimo še -8.

Negativna števila tukaj dobimo z: $N'=2^n-N=\overline{N}+1$

PRIMER 1: $n=4$, N=0100 (+4)

N' =	1	0	0	0	0	
	-	0	1	0	0	
	0	|1	1	0	0	= (-4)

PRIMER 2: $n=4 : N=1001$ (-7)

N' =	1	0	0	0	0	
	-	1	0	0	1	
	0	|0	1	1	1	= (+7)

• Odštevanje izvedemo s seštevanjem in negacijo (dvojiško)

• 2'komplement se danes največ uporablja.

• Seštevanje $N_1 + N_2$:
Podobno kot pri 1'komplementu, samo brez EAC! Tudi če se pojavi, ga ignoriramo.

PRIMER 3: $(+4) + (+3)$:

$$\begin{array}{cccccc} & 0 & 1 & 0 & 0 & \\ + & 0 & 0 & 1 & 1 & \\ \hline & 0 & 1 & 1 & 1 & = +7 \end{array}$$

PRIMER 4: $(+4) + (-3)$:

$$\begin{array}{cccccc} & 0 & 1 & 0 & 0 & \\ + & 1 & 1 & 0 & 1 & \\ \hline \underline{1} & \vert 0 & 0 & 0 & 1 & = +1 \end{array}$$

↑ EAC ignoriramo.

PRIMER 5: $(-4) + (+3)$

$$\begin{array}{cccccc} & 1 & 1 & 0 & 0 & \\ + & 0 & 0 & 1 & 1 & \\ \hline & 1 & 1 & 1 & 1 & = \text{-1} \end{array}$$

PRIMER 6: $(-4) + (-3)$

$$\begin{array}{cccccc} & 1 & 1 & 0 & 0 & \\ + & 1 & 1 & 0 & 1 & \\ \hline \underline{1} & \vert 1 & 0 & 0 & 1 & = \text{-7} \end{array}$$

↑ EAC ignoriramo.

• **Prednosti 2'komplementa:**
- Samo seštevanje in negacije
- 1 ničla
- brez EAC korekcij

• **Zakaj EAC tukaj ni potrebno upoštevati in ga lahko ignoriramo?**

Vzemimo najprej: $-M+N$, $N>M$:

$$-M+N = M'+N = (2^n - M) + N = 2^n + (N-M)$$

Ignoriranje EAC je ekvivalentno odštevanju 2^n, kar daje rezultat $(N-M)$.

Enako velja pri: $-M+(-N)$, $M+N \leq 2^{n-1}$.

$$-M+(-N) = M'+N' = (2^n - M) + (2^n - N) = \underbrace{2^n - (M+N)}_{(M+N)'} + 2^n$$

Če odštejemo 2^n, dobimo ravno $(M+N)'$, kar je v 2'komplementu zapis vrednosti $-(M+N)$.

• OVERFLOW nastopi, ko vsota dveh pozitivnih števil daje negativni rezultat oziroma vsota dveh negativnih števil daje pozitivni rezultat.

Detekcija OVERFLOW-a je mogoča tudi na osnovi "CARRY-IN" in "CARRY-OUT" bitov na najvišjem mestu (prvi je prenos na MSB, drugi pa prenos na MSB + 1). Če velja: $C_{IN} \neq C_{OUT} \rightarrow$ OVERFLOW

PRIMER 7: $5+3$

$\underbrace{c_{OUT}}$	$\underbrace{c_{IN}}$				
0	1	1	1		
	0	1	0	1	(5)
+	0	0	1	1	(3)
	1	0	0	0	(-8) → OVERFLOW

PRIMER 8: $(-7)+(-2)$

$\underbrace{c_{OUT}}$	$\underbrace{c_{IN}}$				
1	0	0	0		
	1	0	0	1	(-7)
+	1	1	1	0	(-2)
1|	0	1	1	1	(+7) → OVERFLOW

PRIMER 9: $5+2$

$\underbrace{c_{OUT}}$	$\underbrace{c_{IN}}$				
0	0	0	0		
	0	1	0	1	(+5)
+	0	0	1	0	(+2)
	0	1	1	1	(+7) → NI OVERFLOW

Povzetek lastnosti vseh treh predstavitev predznačenih števil podaja tabela:

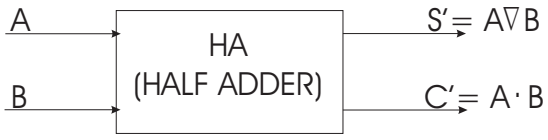
	Predznak-veličinski	1'komplement	2'komplement
št. ničel	2	2	1
št. operacij	4 - seštevanje - odštevanje - test predznaka - test veličine	3 - seštevanje - test predznaka - negacija	2 - seštevanje - negacija
korekcija EAC	NE	DA	NE

Poglavje 3

STRUKTURE BINARNIH OPERACIJ

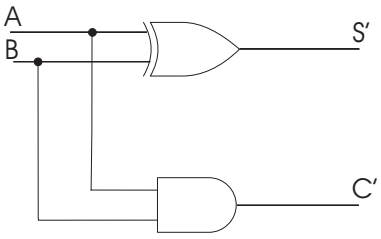
3.1 Seštevanje

3.1.1 Polovični seštevalnik



Slika 3.1: Polovični seštevalnik

Na sliki 3.2 vidimo vezje polovičnega seštevalnika.



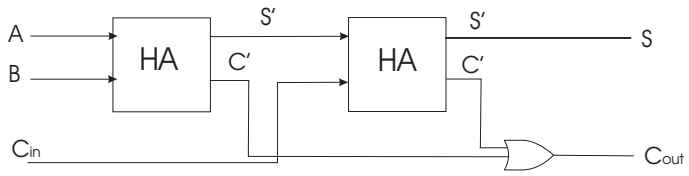
Slika 3.2: Vezje polovičnega seštevalnika

Pravilnostna tabela za polovični seštevalnik:

A	B	S'	C'
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

3.1.2 Polni seštevalnik

Velja za vsako bitno pozicijo. Vidimo ga na sliki 3.3.

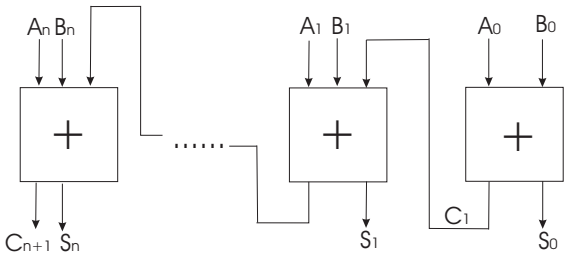


Slika 3.3: Polni seštevalnik

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$S = (A \bigtriangledown B) \bigtriangledown C_{in} = A \bigtriangledown B \bigtriangledown C_{in}$
$C_{out} = (A \bigtriangledown B) \cdot C_{in} \vee A \cdot B = (A \vee B)C_{in} \vee A \cdot B$

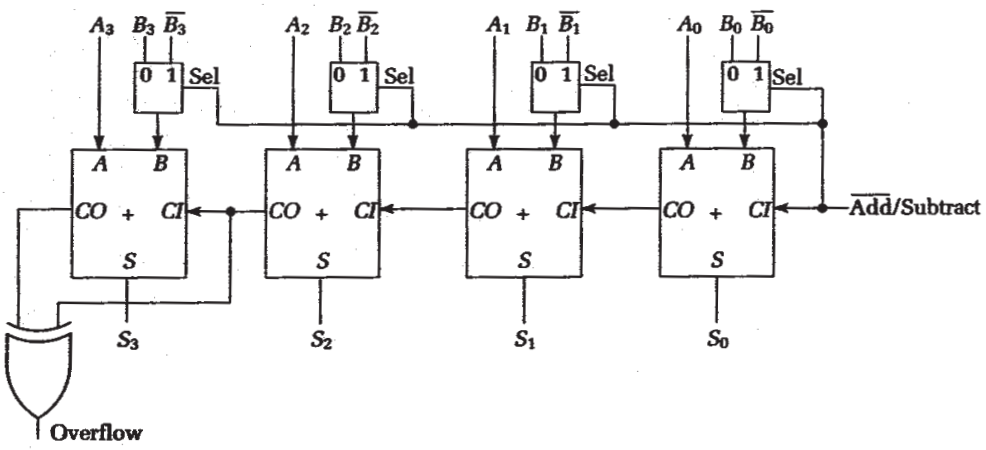
Na sliki 3.4 je vezava polnih seštevalnikov v kaskado. Tako dobimo večbitbni seštevalnik.



Slika 3.4: Kaskadna vezava polnih seštevalnikov

3.1.3 Seštevalnik/odštevalnik

S pomočjo polnega seštevalnika in MX vezij sestavimo seštevalnik/odštevalnik, ki ga vidimo na sliki 3.5. Pri izboru SUBTRACT vstopa "1" na pozicijo 0, kar



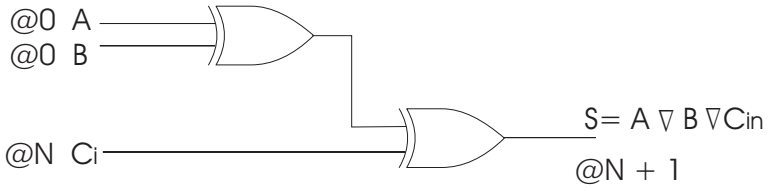
Slika 3.5: Seštevalnik / odštevalnik

pomeni da k $A+\overline{B}$ dodamo 1, ki določa 2'komplement. EX-OR določa OVERFLOW, kadar sta zadnja prenosa (C_{in}, C_{out}) različna (glej poglavje 2'komplement in primere).

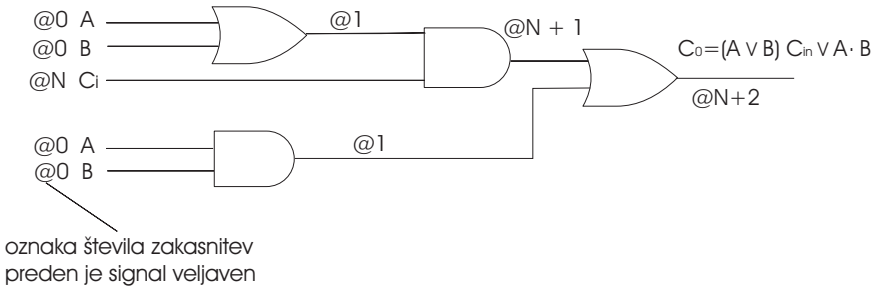
3.1.4 "CARRY LOOK AHEAD" vezja (CLA)

Imajo za cilj, da reducirajo čas za izvedbo operacije seštevanja.

• Analiza kritičnih poti
Gre predvsem za signale prenosa, ki se serijsko prenašajo od najnižje bitne pozicije do najvišje. Predpostavljali bomo enake zakasnitve (1@) za vsa vrata, kar v praksi ni res.

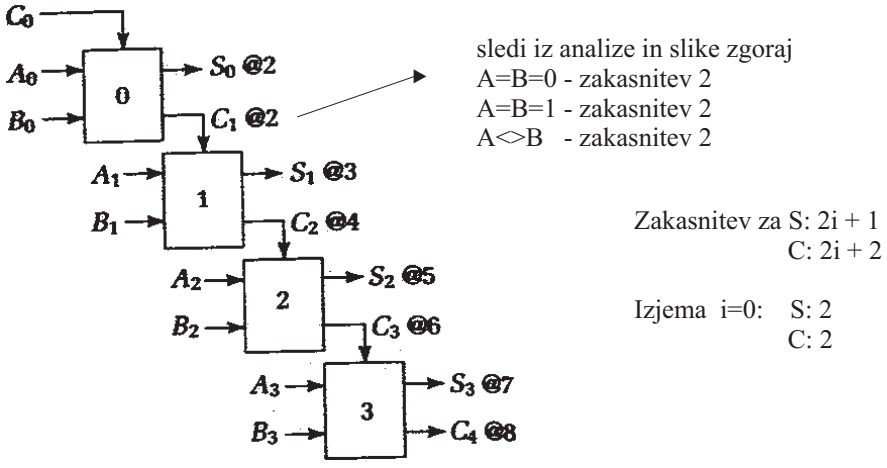


Slika 3.6: Vsota.



Slika 3.7: Prenos.

Časovno najzahtevnejšo kombinacijo predstavlja seštevanje $1111_2 + 0001_2$ (n=4), ker vsak bit generira prenos. Razmere prikazuje slika 3.8.



Slika 3.8: Najzahtevnejša kombinacija seštevanja

V splošnem je zakasnitev: $2 \cdot \mathrm{i} + 1$ za S_i ($\mathrm{i} > 0$) in $2 \cdot \mathrm{i} + 2$ za C_o (če je $\mathrm{i} = 0$ $\rightarrow$ zakasnitev je 2 za S in C)

• Bistvo pohitritve signalov prenosa je v direktnem izražanju C_i v odvisnosti od $A_i, A_{i-1}, \ldots, A_0$ in $B_i, B_{i-1}, \ldots, B_0$ in C_0. S tem postane funkcija bolj kompleksna, vendar jo lahko realiziramo 2-nivojsko.

Vpeljali bomo 2 novi funkciji, s pomočjo katerih bomo konstruirali prenosno funkcijo.

$G_i = A_i \cdot B_i$ - Carry Generate
$P_i = A_i \bigtriangledown B_i$ - Carry Propagate

S pomočjo pomožnih funkcij G_i in P_i lahko izrazimo S in C_{out} (vsoto in izhodni prenos):

$S_i = A_i \bigtriangledown B_i \bigtriangledown C_i = P_i \bigtriangledown C_i$
$C_{i+1} = A_iB_i + A_iC_i + B_iC_i = A_iB_i + C_i(A_i \bigtriangledown B_i) = G_i + C_iP_i$

Sedaj lahko prenosno logiko za posamezne bitne pozicije pišemo drugače:

$C_1 = G_0 \vee P_0 \cdot C_0$
$C_2 = G_1 \vee P_1 \cdot C_1 = G_1 \vee P_1G_0 \vee P_1P_0C_0$
$C_3 = G_2 \vee P_2 \cdot C_2 = G_2 \vee P_2G_1 \vee P_2P_1G_0 \vee P_2P_1P_0C_0$
$C_4 = G_3 \vee P_3 \cdot C_3 = G_3 \vee P_3G_2 \vee P_3P_2G_1 \vee P_3P_2P_1G_0 \vee P_3P_2P_1P_0C_0$

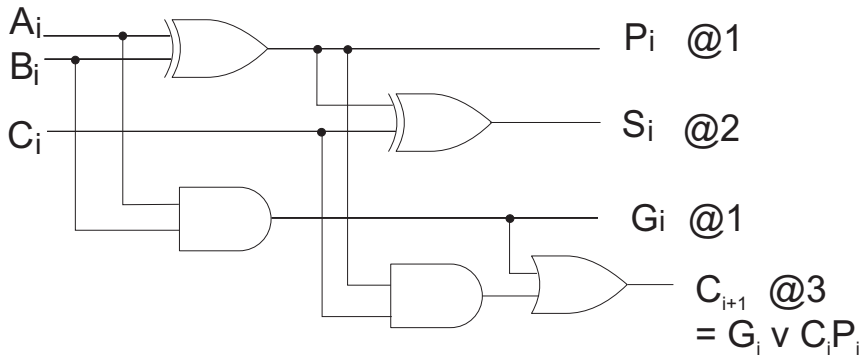
V 2-nivojski obliki ima C_i $(i + 1)$ konjunkcij, od katerih ima najdaljša $i + 1$ vhodov. To postavlja praktične omejitve glede števila stopenj, preko katerih se računa prenos. Standardna vezja realizirajo prenos preko 4 stopenj.

• Implementacija "Carry-Lookahead" logike: seštevalna stopnja z izhodi P, G in S (slika 3.9).

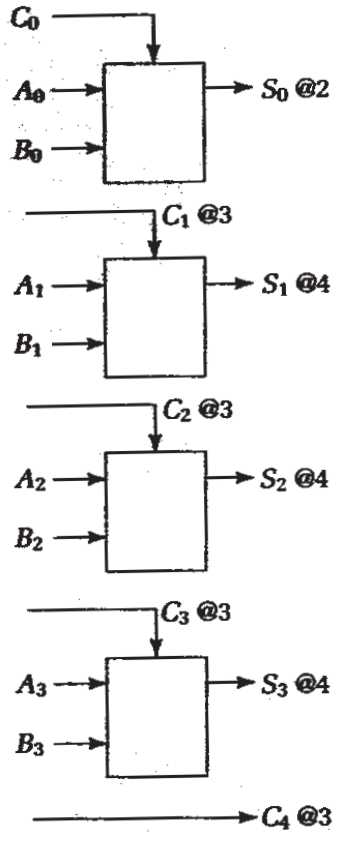
C_i ($\mathrm{i} = 1 \div 4$) realiziramo 2-nivojsko na osnovi gornjih izrazov. Če imamo dostopne P_i in G_i, izračunamo C_i z 2 dodatnima zakasnitvama vrat. Za vsote S_i pa rabimo še eno zakasnitev. Razmere podaja slika 3.10.

• Zaradi omenjenih omejitev (fan-in) uporabljamo pri širših seštevalnikih hierarhičen pristop.

16-bitni seštevalnik implementiramo s štirimi 4-bitnimi seštevalniki (z vgrajeno



Slika 3.9: Seštevalna stopnja.



Slika 3.10:

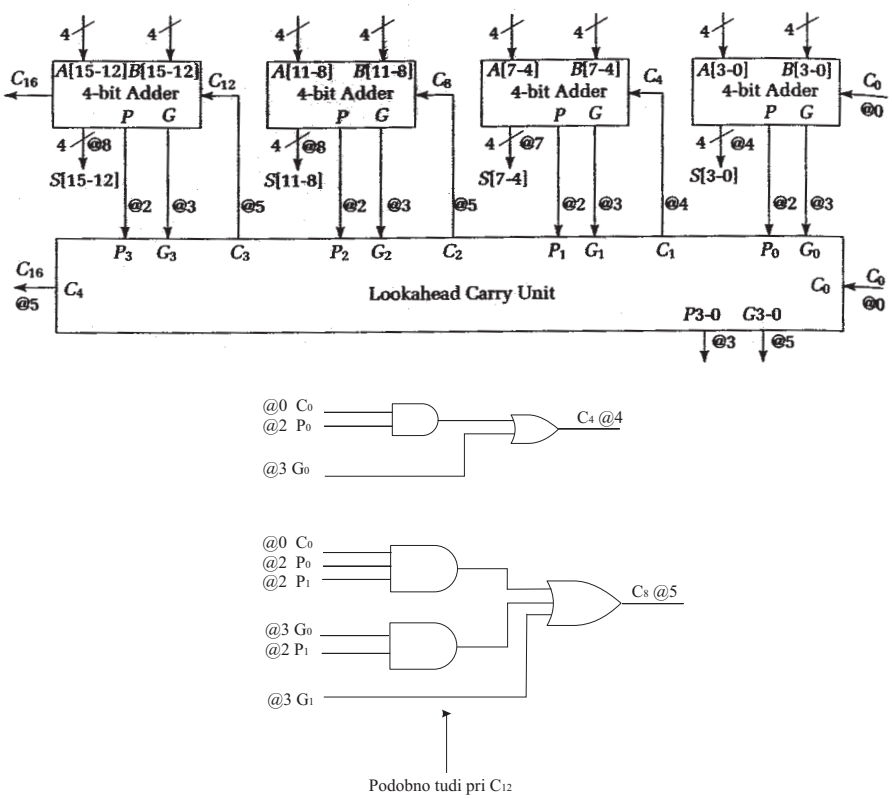
4-bitno CLA logiko). Vsak 4-bitni seštevalnik izračuna tudi svojo "grupno" P

in G funkcijo:

$$P = P_3P_2P_1P_0$$
$$G = G_3 \vee G_2P_3 \vee G_1P_3P_2 \vee G_0P_3P_2P_1$$

Vezje takega seštevalnika vidimo na sliki 3.11. Vezje drugega nivoja izračuna prenose med 4-bitnimi seštevalniki in izhodni prenos. Logika je identična kot pri izračunu prenosov znotraj 4-bitnih seštevalnikov (glej enačbe):

$$\begin{aligned} P_0 &= P_1^0P_2^0P_3^0P_4^0 \\ P_i^0 &= A_i \bigtriangledown B_i \\ G_0 &= G_3^0 \vee G_2^0P_3^0 \vee G_1^0P_3^0P_2^0 \vee G_0^0P_3^0P_2^0P_1^0 \\ C_4 &= G_0 \vee C_0P_0 \\ C_8 &= G_1 \vee G_0P_1 \vee P_1P_0C_0 \end{aligned}$$



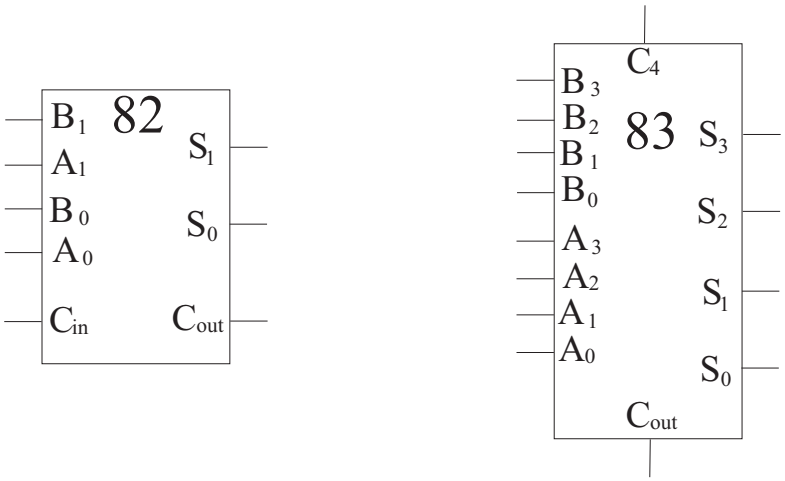
Slika 3.11: 16-bitni seštevalnik, implementiran s štirimi 4-bitnimi seštevalniki

V 8 zakasnitvah lahko izračunamo 16-bitno vsoto (proti 32 zakasnitvam brez CLA). 64-bitni seštevalnik bi lahko podobno sestavili iz štirih 16-bitnih , itd.

• TTL komponente za seštevalnik in CLA:

- 7482: 2 FA (Full Adder) z "ripple" prenosom (slika 3.12)

- 7483: 4-bitni FA s CLA (slika 3.12)



Slika 3.12: 7482: 2-bitni polni seštevalnik z 'ripple' prenosom. 7483: 4-bitni polni seštevalnik (s CLA)

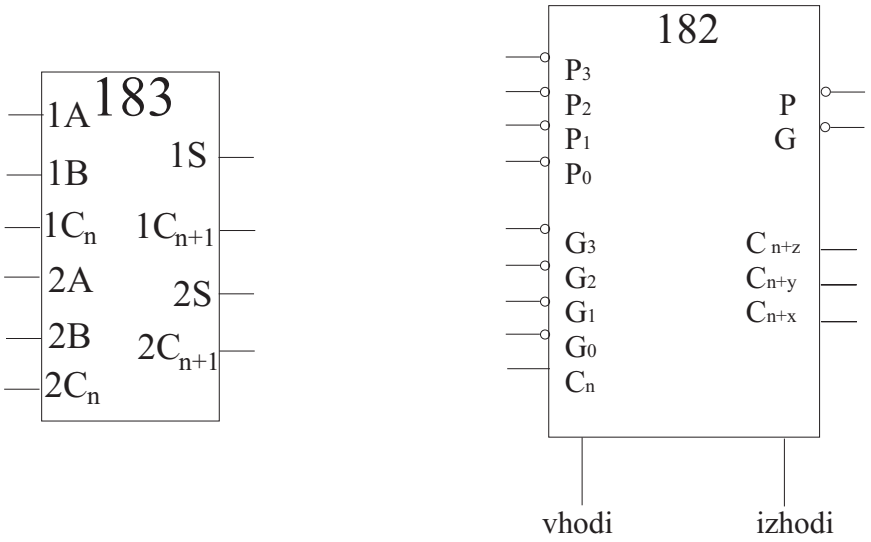
- 74183: 2 samostojna FA (1 bitna) (slika 3.13)

- 74182: 4-bitna CLA enota: Posebno razvita v kombinaciji s 74181 ALU enoto (slika 3.13)

3.2 Aritmetično logična enota (ALU)

Vsebuje kombinacijska vezja, ki implementirajo logične ali aritmetične operacije.

• n-bitni ALU ima 2 vhodni besedi, A= $A_{n-1}, \ldots, A_0$ in B = $B_{n-1}, \ldots, B_0$, ter izhodno besedo $F = F_n, F_{n-1}, \ldots, F_0$, pri čemer je F_n dejansko izhodni prenos, C_{out}. Dodatni vhod je še vhodni prenos oziroma C_{in}.



Slika 3.13: 74183: 2 samostojna polna seštevalnika, 74182: 4-bitna CLA enota.

Kontrolni vhodi določajo tip operacije (M = 0 določa logično, M = 1 pa aritmetično operacijo) in operacijo (vhodi S_i).

• 1-bitna rezina ALU enote je v tabeli 3.2.

POZOR: $\overline{A} \equiv$ 1'komplement od A, $A' = \overline{A} + 1 \equiv$ 2'komplement

•

S kaskado rezin dobimo n-bitno ALU enoto. Pri tem predstavlja omejitev prenos med rezinami in njihova zakasnitev. Glede na specifikacijo ALU rezine (zgornja tabela) izgleda rezina tako, kot kaže slika 3.14.

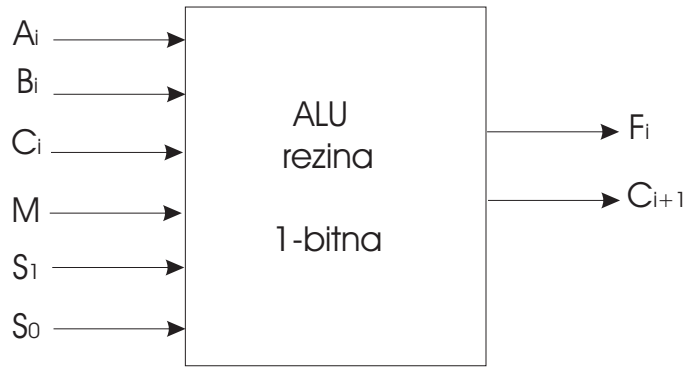
Ustrezna pravilnostna tabela izkazuje veliko redundanc. V postopku minimizacije bi lahko dobili 2-nivojsko implementacijo, npr. s programom ESPRESSO rešitev, ki za F_i zahteva kar 18 konjunkcij in za c_{i+1} 5. Program MIS II najde večnivojsko rešitev z 11 vrati in 1 EX-OR za oba izhoda plus 5 invertorjev.

Z ročnim minimiziranjem pa dobimo najboljšo večnivojsko rešitev. Upoštevali smo dejstvo, da $S_1 = 0$ blokira B_i glede na izhod s pomočjo vrat A_i (in še več podobnih detajlov), ki jih vidimo na sliki 3.15).

• Najbolj razširjena ALU enota v TTL je 74181 (slika 3.16). Omogoča računanje

S_1	S_0	Funkcija	Komentar
		M = 0 (logične operacije)	
0	0	$F_i = A_i$	
0	1	$F_i = \overline{A_i}$	
1	0	$F_i = A_i \bigtriangledown B_i$	
1	1	$F_i = \overline{A_i \bigtriangledown B_i}$	
		M = 1, $C_0 = 0$ (Aritmetične operacije)	
0	0	$F_i = A$	
0	1	$F_i = \overline{A}$	
1	0	$F_i = A + B$	
1	1	$F_i = \overline{A} + B$	Vsota B in $\overline{A}$ (ni preveč uporabno)
		M=1, $C_0 = 1$ (Aritmetične operacije)	
0	0	$F_i = A + 1$	
0	1	$F_i = \overline{A} + 1$	2' komplement od A
1	0	$F_i = A + B + 1$	(A+B) +1
1	1	$F_i = \overline{A} + B + 1$	B - A

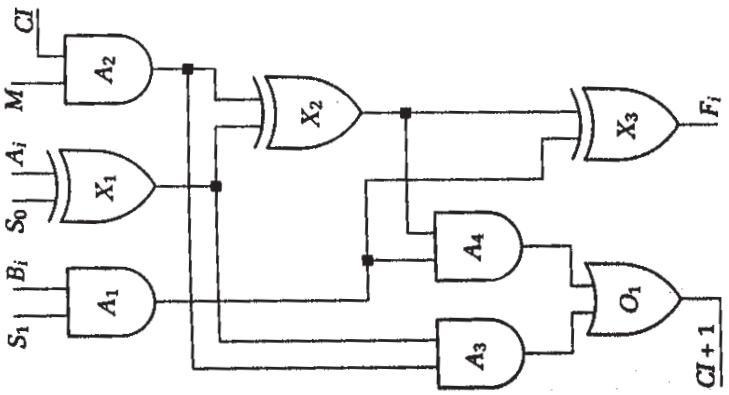
Tabela 3.1: 1-bitna rezina ALU enote.



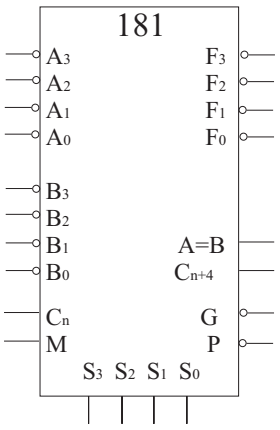
Slika 3.14: ALU rezina

32 različnih operacij (16 logičnih in 16 aritmetičnih (s $C_n = 0$ in $C_n = 1$)).
Deluje tudi kot komparator (A=B).

Logične operacije: NAND, NOR, AND, OR, XOR, XNOR,...
Aritmetične operacije: +, -,...
Vsebuje logiko za CLA (4-bitni) in daje na izhodu P in G za povezovanje elementov v večjo enoto. Slika 3.17 prikazuje delovanje ALE 74181.



Slika 3.15:



Slika 3.16: 74181

Podatki na vhodu in izhodu so v NEGATIVNI logiki. Vezje lahko uporabimo tudi s podatki v POZITIVNI logiki. Tedaj moramo pri aritmetičnih operacijah komplementirati C_n (namesto $C_n = 0 \rightarrow \overline{C_n} = 1$ in namesto $C_n = 1 \rightarrow \overline{C_n} = 0$) in tudi C_{n+4}.

• 74181 in 74182 omogočata konstruiranje 16-bitne ALU enote, ki jo vidimo na sliki 3.18.

Selection				$M = 1$	$M = 0$, Arithmetic Functions	
S_3	S_2	S_1	S_0	Logic Function	$C_n = 0$	$C_n = 1$
0	0	0	0	$F =$ not A	$F = A$ minus 1	$F = A$
0	0	0	1	$F = A$ nand B	$F = AB$ minus 1	$F = AB$
0	0	1	0	$F =$ (not A) $\vee B$	$F = A$ (not B) minus 1	$F = A$ (not B)
0	0	1	1	$F = 1$	$F =$ minus 1	$F =$ zero
0	1	0	0	$F = A$ nor B	$F = A$ plus ($A \vee$ not B)	$F = A$ plus ($A \vee$ not B) plus 1
0	1	0	1	$F =$ not B	$F = AB$ plus ($A \vee$ not B)	$F = AB$ plus ($A \vee$ not B) plus 1
0	1	1	0	$F = A$ xnor B	$F = A$ minus B minus 1	$F =$ ($A \vee$ not B) plus 1
0	1	1	1	$F = A \vee$ not B	$F = A \vee$ not B	$F = A$ minus B
1	0	0	0	$F =$ (not A) B	$F = A$ plus ($A \vee B$)	$F = A$ plus ($A \vee B$) plus 1
1	0	0	1	$F = A$ xor B	$F = A$ plus B	$F = A$ plus B plus 1
1	0	1	0	$F = B$	$F = A$ (not B) plus ($A \vee B$)	$F = A$ (not B) plus ($A \vee B$) plus 1
1	0	1	1	$F = A \vee B$	$F =$ ($A \vee B$)	$F =$ ($A \vee B$) plus 1
1	1	0	0	$F = 0$	$F = A$	$F = A$ plus 1
1	1	0	1	$F = A$ (not B)	$F = AB$ plus A	$F = AB$ plus A plus 1
1	1	1	0	$F = AB$	$F = A$ (not B) plus A	$F = A$ (not B) plus A plus 1
1	1	1	1	$F = A$	$F = A$	$F = A$ plus 1

Slika 3.17: Delovanje ALE 74181.

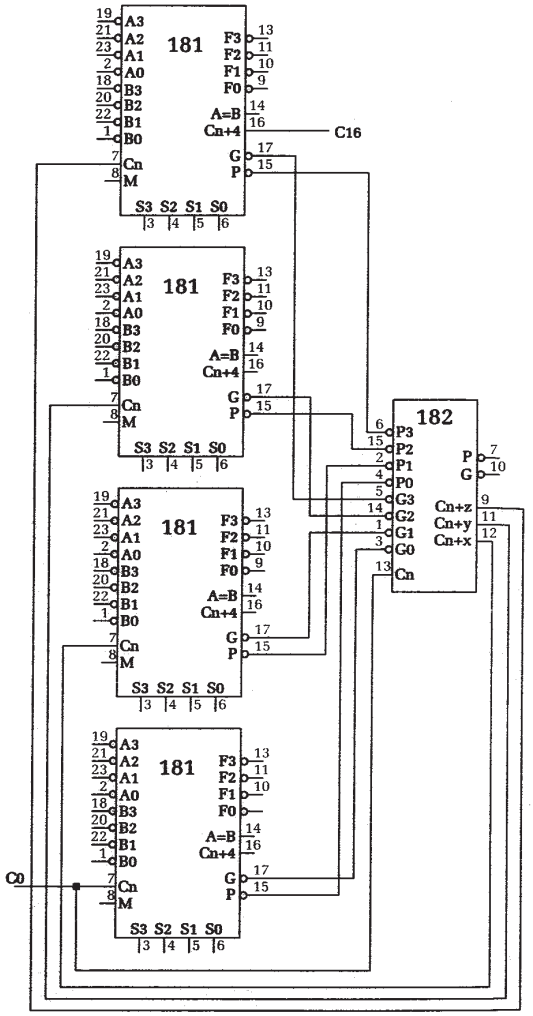
3.3 BCD seštevanje ("Binary Coded Decimal")

Danes se uporablja le zaradi kompatibilnosti s starimi stroji, včasih pa zaradi enostavne pretvorbe v alfanumerično predstavitev za izpise in prikaze na zaslonu.

x_1	x_2	x_3	x_4	BCD	
0	0	0	0	0	
0	0	0	1	1	
	⋮			⋮	
1	0	0	1	9	
1	0	1	0	X	6 kombinacij je prepovedanih (X)
	⋮			⋮	
1	1	1	1	X	

Osnovni princip je enak kot pri binarnem seštevanju, le prenose je potrebno prirediti za BCD kodo. Če vsota prekorači 9, je potrebno dodati k njej še 6.

PRIMER:



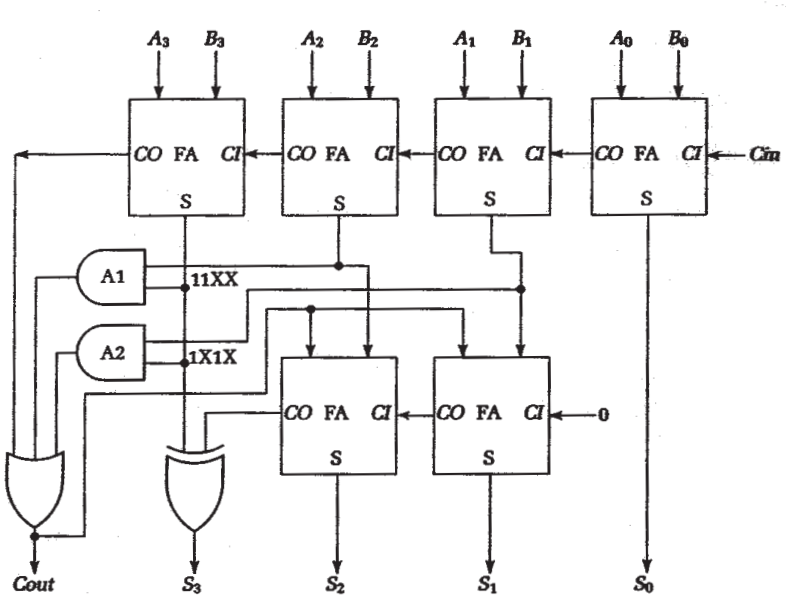
Slika 3.18: 16 bitna ALU enota

0101	(5)
1000	(8)
1101	= 13 v dec. sistemu
+ 0110	(6)
1 0011	= 13 v BCD sistemu

Še drugi primer:

1001	(9)
0111	(7)
10000	= 16 v dec. sistemu
+ 0110	(6)
1 0110	= 16 v BCD sistemu

• Implementacija BCD seštevanja:
Na sliki 3.19 prva vrstica realizira 4-bitni binarni seštevalnik, druga pa doda 0110_2, kadar je vsota v prvi vrstici večja od 1001_2. Vrata A_1 in A_2 določata



Slika 3.19: Implementacija BCD seštevanja.

pogoje, ko vsota v 1. vrstici presega $(1001)_2$. To sta ravno $11XX_2$ in $1X1X_2$, dodaten pogoj pa je C_0 iz prve vrstice (drugi primer zgoraj).

	1100		1010
11XX =	1101	1X1X =	1011
	1110		1110
	1111		1111

Dodaten razlog, da se BCD seštevanje opušča, je v precej obsežnejšem "hardware"-u (več kot 50 % preko binarnega seštevalnika).

3.4 Kombinacijski množilniki

• Kombinacijski množilniki so hitrejši od sekvenčnih

• Ogledali si bomo množenje veličinskih delov, predznak je enostavno določljiv z XOR operacijo nad predznaki obeh faktorjev.

PRIMER:

				1	1	0	1	(13) - multiplikand (množenec)
			*	1	0	1	1	(11) - multiplikator (množitelj)
				1	1	0	1	Vmesni produkt
			1	1	0	1		Vmesni produkt
		0	0	0	0			Vmesni produkt
	1	1	0	1				Vmesni produkt
1	0	0	0	1	1	1	1	(143) - Končni rezultat

Vsak bit multiplikatorja (množitelja) določa zamik multiplikanda (množenca) v levo. Vsota zamaknjenih multiplikandov (toliko kot enic v multiplikatorju) je enaka produktu. Zamaknjenim multiplikandom pravimo tudi vmesni produkti.

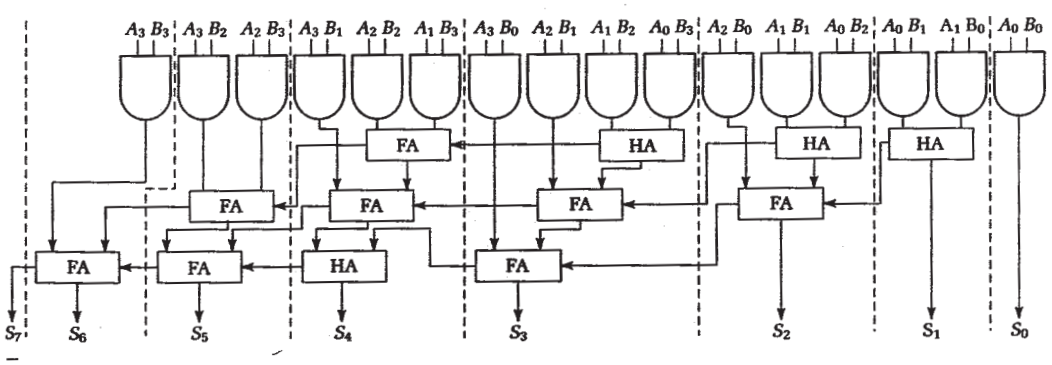
• Množenje dveh n-bitnih števil daje rezultat dolžine 2n. ($2^n * 2^n = 2^{2n}$).

• Kombinacijsko vezje direktno implementira proces seštevanja vmesnih produktov. Splošno predstavimo množenje dveh števil A in B z:

			A_3	A_2	A_1	A_0	
			B_3	B_2	B_1	B_0	
			A_3B_0	A_2B_0	A_1B_0	A_0B_0	
		A_3B_1	A_2B_1	A_1B_1	A_0B_1		
	A_3B_2	A_2B_2	A_1B_2	A_0B_2			A_XB_X so parcialni produkti
A_3B_3	A_2B_3	A_1B_3	A_0B_3				
S_6	S_5	S_4	S_3	S_2	S_1	S_0	

Vsaka konjunkcija se imenuje parcialni produkt. Rezultat dobimo s seštevanjem parcialnih produktov v posameznih stolpcih. Pri tem prenašamo "Carry" od desne proti levi.

• Kombinacijsko vezje za realizacijo 4-bitnega množenja na osnovi seštevanja parcialnih produktov v stolpcih vmesnih produktov prikazuje slika 3.20.



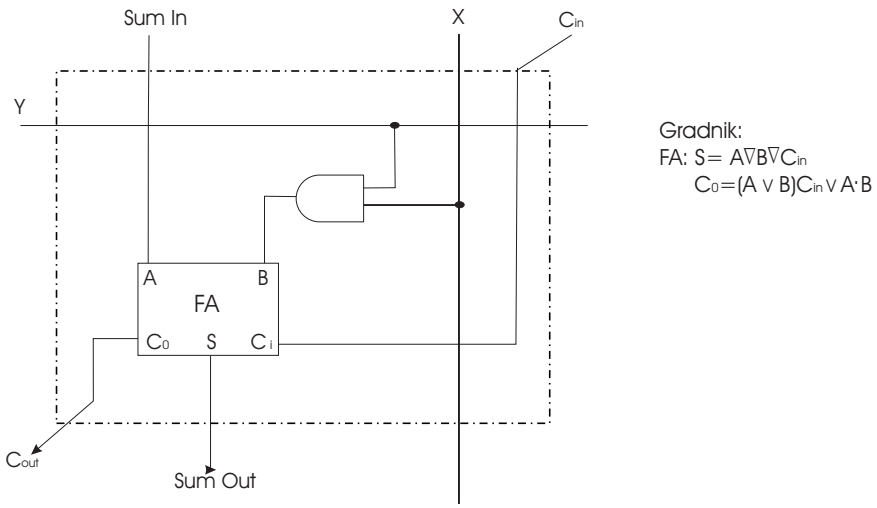
Slika 3.20: Množilnik 4 x 4

3.4.1 Množilnik 4 x 4

Zakasnitev vezja določajo prenosi med seštevalniki. S CLA lahko zmanjšamo te zakasnitve. Vezje ima 4 HA in 8 FA. Zaradi lažje implementacije lahko vzamemo samo FA in tam, kjer ni potrebno, postavimo C_i na 0.

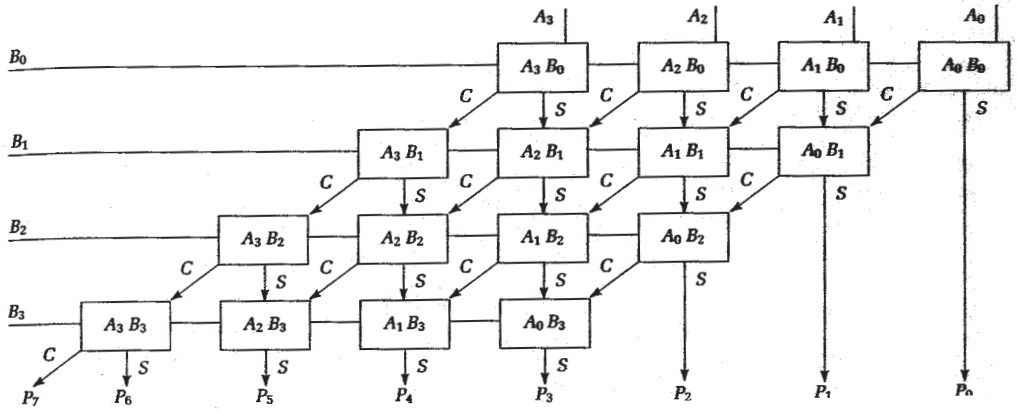
• 1 FA vsebuje 6 vrat, 12 FA torej vsebuje 12 * 6 = 72 vrat, plus 16 vrat za parcialne produkte, kar da skupaj 88 vrat.

• Bolj homogena realizacija z enim samim gradnikom pa je prikazana skupaj s strukturo gradnika na sliki(3.21).



Slika 3.21: Struktura gradnika.

4-bitno množenje vidimo na sliki 3.22. Pri tem se A_i vrednosti prenašajo po diagonalah, B_i vrednosti pa po horizontalah - ti prenosi niso označeni eksplicitno.



Slika 3.22: 4-bitno množenje.

3.4.2 Množilnik 8 x 8

Privzeli bomo množilnik 4x4 kot osnovni blok in pokazali, kako lahko zgradimo množilnik 8x8. Poleg tega bloka bomo uporabili še seštevalnik, ALU in CLA.

• Princip, kot smo ga uporabili pri 4x4 množilniku, bomo ponovili, le da bodo tukaj parcialni produkti produkti dveh 4-bitnih števil

$A_{7-0} * B_{7-0} = P_{15-0}$.

A in B razdelimo v 2 grupi po 4 bite:

		A_{7-4}	A_{3-0}	
	$*$	B_{7-4}	B_{3-0}	
		A_{3-0} $*$	B_{3-0}	$= PP_0$ (parcialni produkt 0)
	A_{7-4} $*$	B_{3-0}		$= PP_1$ (parcialni produkt 1)
	A_{3-0} $*$	B_{7-4}		$= PP_2$ (parcialni produkt 2)
A_{7-4} $*$	B_{7-4}			$= PP_3$ (parcialni produkt 3)
P_{15-12}	P_{11-8}	P_{7-4}	P_{3-0}	

PRIMER:

		1111	0010	(242)
	*	1000	1100	(140)
		0001	1000	= 0010 * 1100 = $PP0$
	1011	0100		= 1111 * 1100 = $PP1$
	0001	0000		= 0010 * 1000 = $PP2$
0111	1000			= 1111 * 1000 = $PP3$
1000	0100	0101	1000	(33880)

Končni produkt dobimo z vsoto parcialnih produktov $PP_0 \div PP_3$:

$$\begin{aligned} P_{3-0} &= PP0_{3-0} \\ P_{7-4} &= PP0_{7-4} + PP1_{3-0} + PP2_{3-0} \\ P_{11-8} &= PP1_{7-4} + PP2_{7-4} + PP3_{3-0} \\ P_{15-12} &= PP3_{7-4} \end{aligned}$$

Seveda moramo pri seštevanju upoštevati prenose.

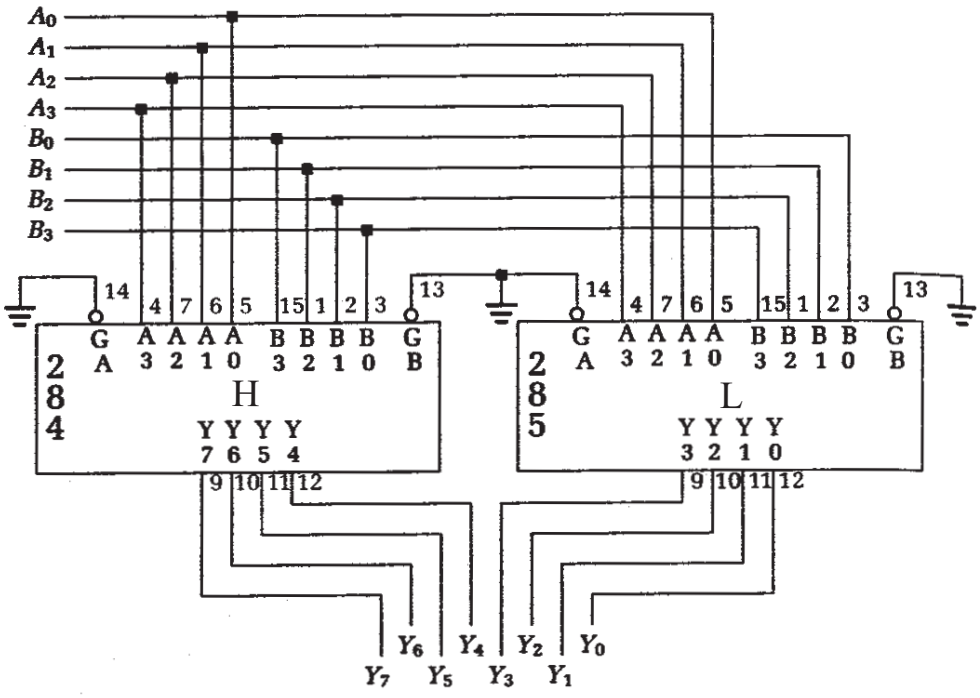
• Implementacija množilnika 8x8. Sestavljen je iz naslednjih osnovnih blokov:

1. blok za izračun parcialnih produktov
2. seštevanje 4 bitnih rezin
3. CLA

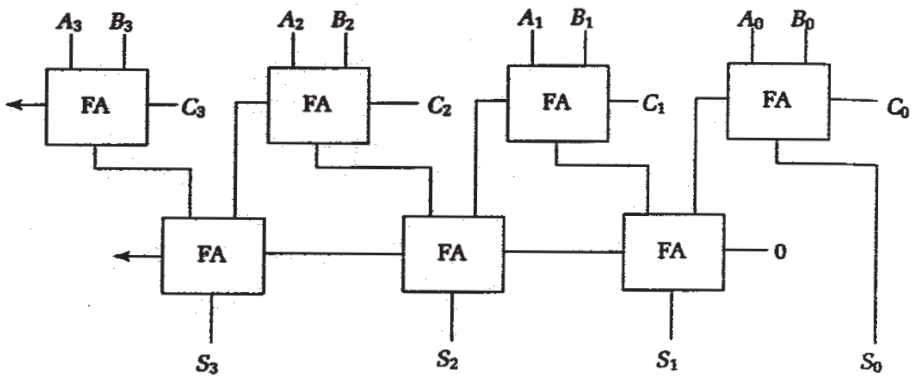
Izračun parcialnih produktov:
Vsak od 8-bitnih parcialnih produktov je implementiran z 74284/74285 parom (glej sliko 3.23).

74284 realizira zgornje 4 bite produkta, 74285 pa spodnje 4 bite. $\overline{G_A}$ in $\overline{G_B}$ sta "active-low enable" bita in morata biti 0, sicer so izhodi v visoko-impedačnem stanju.

Izračun vsot:
Spodnje 4 bite rezultata (P_{3-0}) dobimo brez seštevanja ($P_{3-0} = PP0_{3-0}$). P_{7-4} in P_{11-8} pa zahtevata vsoto treh 4-bitnih števil. Za to potrebujemo sledeče vezje (slika 3.24).

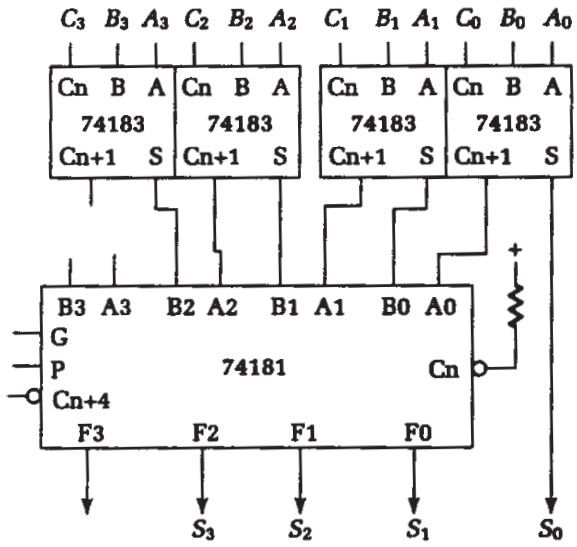


Slika 3.23: Izračun parcialnih produktov.



Slika 3.24: Izračun vsot $(A + B + C)$.

Seštevamo $A_{3-0} + B_{3-0} + C_{3-0}$ in vsi vhodi vstopajo v 1.nivo. 2.nivo združuje prenose in sume prejšnjega nivoja. To blok shemo realiziramo z dvema 74183 in enim 74181 (ALU). Realizacijo vidimo na sliki 3.25



Slika 3.25: 2 x 74183 in 74181 ALU.

CLA:
Ker gre za izračun treh parcialnih produktov (P_{3-0} ne zahteva računanja), potrebujemo 3x74181 in za pohitritev lahko med njimi prenose vežemo na enoto 74182 (CLA). To vezje predstavlja slika 3.26.

• Podrobna analiza, ki jo izpuščamo, pokaže, da je tipična zakasnitev množilnika 8x8 103 ns, maksimalna ("worst-case") pa 171 ns.

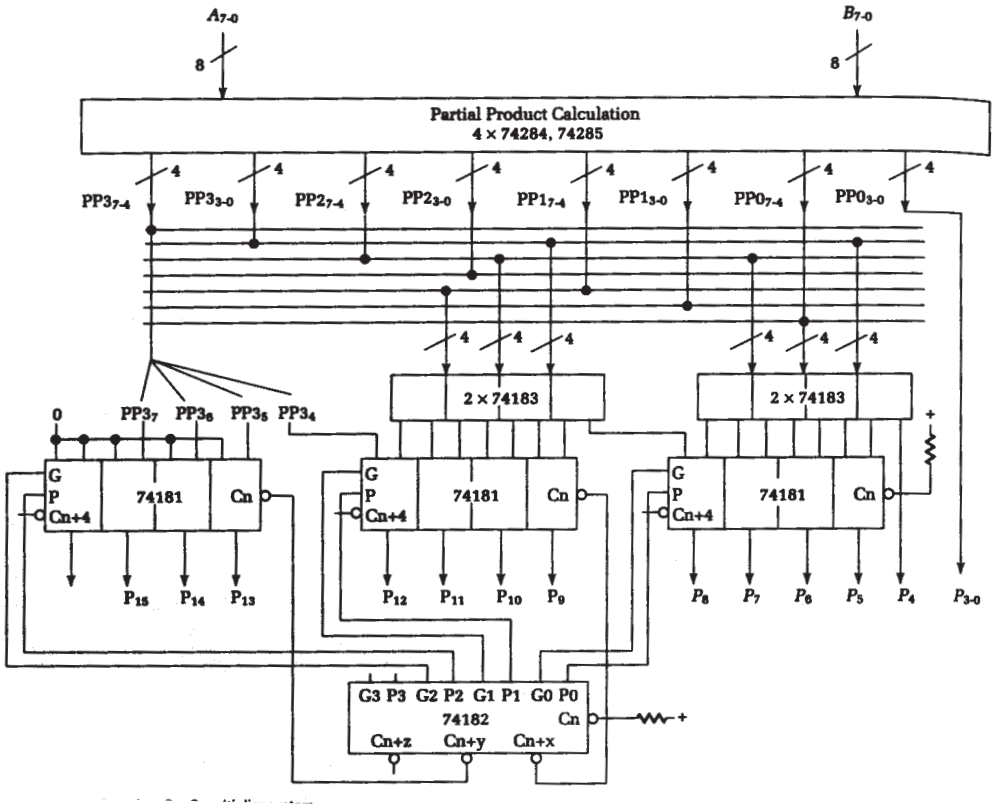
3.5 Binarno deljenje

• Ogledali si bomo binarno deljenje veličin A in B (brez predznakov)

$A = A_n A_{n-1} \ldots A_0$
$B = B_n B_{n-1} \ldots B_0$

$A : B = C_n C_{n-1} \ldots C_0$, ostanek R

• Postopek:



Slika 3.26: Celotno vezje množilnika 8 x 8.

1. i=n, R = A
2. če R - 2^i B $\geq 0 \rightarrow C_i = 1$ in odštej 2^i B od R. Če R - 2^i B $< 0 \rightarrow C_i = 0$
3. Ponovi 2. korak za i=n-1, ...,1,0 tako, da dobimo C_{n-1} ..., C_1, C_0. Na koncu je rezultat A:B= $C_n C_{n-1} \ldots C_1 C_0$, ostanek = R.

PRIMER (n=4):
A = 11101 (29)
B = 00101 (5)

1. i = 4, R = 11101
   R - 2^4 B = 11101 - 001010000 $< 0 \rightarrow C_4 = 0$
2. i = 3, R - 2^3 B = 11101 - 00101000 $< 0 \rightarrow C_3 = 0$

3. i = 2, R - 2^2 B = 11101 - 0010100 $> 0 \rightarrow C_2 = 1$
   R = R - 2^2 B = 11101 - 10100 = 01001

4. i = 1, R - 2^1 B = 01001 - 1010 = 00100 $< 0 \rightarrow C_1 = 0$

5. i = 0, R - 2^0 B = 01001 - 0101 = 00100 $> 0 \rightarrow C_0 = 1$

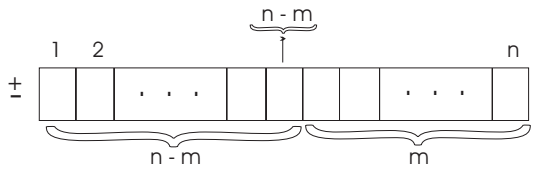
   A : B = 00101 (5), ostanek = 00100 (4)

Tabelarično:

	B	
	00101	
00101	11101	A
↑ C	- 10100	(2^2 B)
	01001	
	- 00101	(2^0 B)
	00100	= R

3.6 Aritmetika plavajoče vejice

• Do sedaj smo obravnavali zapis števil v fiksni vejici, kjer lahko število cifer za zapis določenih razponov števil hitro naraste. Tukaj imamo v splošnem:



V zgornjem formatu fiksne vejice lahko zapišemo obseg:

$$2^{-m} \leq N \leq (2^{n-m} - 1 + \sum_{i=1}^{m} 2^{-i})$$

oziroma

$$2^{-m} \leq N \leq 2^{n-m} - 2^{-m}$$

• Plavajoča vejica omogoča zapis števil z manj ciframi, npr.: $5000000 = 5 \cdot 10^6 = 0,5 \cdot 10^7$.

Splošen zapis števil v plavajoči vejici:
$N = c^a \cdot A = M \cdot 2^E, \quad \frac{1}{2} \leq M < 1.$

Obstajata dva formata v plavajoči vejici:

A. Znanstveni format: $(-1)^S \times F \times 2^E$, kjer je F definiran z $1/2 \leq F < 1$.
Npr., $-0,11 \rightarrow (-1)^1 \times (0,11) \times 2^0$

B. Normaliziran format: $(-1)^S \times (1+F) \times 2^E$, kjer je F definiran z $0 \leq F < 1$.
Npr., $-0,11 \rightarrow (-1)^1 \times (1,10) \times 2^{-1}$

PRIMER:

$$\begin{array}{lcl} 1011000 & = & (.1011) \cdot 2^{111} \\ 0{,}00001 & = & (.1) \cdot 2^{-100} \end{array}$$

$N_2 = M \cdot 2^E = \pm 0, A_1 A_2 \ldots A_m \cdot 2^{\pm a_1 \ldots a_n}$

• Operacije v plavajoči vejici:
- množenje:
$x = M_x \cdot 2^{E_x}$; $y = M_y \cdot 2^{E_y}$
$x \cdot y = (M_x \cdot M_y) \cdot 2^{E_x + E_y}$

- deljenje:
$x : y = (M_x : M_y) \cdot 2^{E_x - E_y}$

Če $(M_x \cdot M_y)$ oziroma $(M_x : M_y)$ izpadeta iz omejitve $(\frac{1}{2} \leq M < 1)$, ju moramo normirati tako, da spremenimo eksponent, npr.: $(.1) \cdot 2^1 \cdot (.1) \cdot 2^1 = (.01) \cdot 2^2 = (.1) \cdot 2^1$

- Seštevanje/odštevanje:
Števila morajo imeti enake eksponente.
$(.1) \cdot 2^9 + (.1) \cdot 2^{10} = (.01) \cdot 2^{10} + (.1) \cdot 2^{10} = (.01 + .1) \cdot 2^{10} = (.11) \cdot 2^{10}$
oziroma splošno: $x + y = (M_x + M_y) \cdot 2^{E_x}$

Poglavje 4

SINTEZA VEČNIVOJSKE LOGIKE

• Pri dvonivojski logiki, ki je hitra, včasih pride do zahtev po velikem številu vhodov v operatorje (velik "fan-in"). Tedaj moramo preiti na večnivojsko logiko.

• Pri večnivojski logiki gre za odločitev ("trade-off") med časom (zakasnitev z nivoji narašča) in obsegom vrat oziroma prostorom. Praviloma večnivojske rešitve zahtevajo manj vrat, so pa počasnejše.

• Reševanje večnivojske logike je bolj zahtevno kot pri dvonivojski logiki. Tudi definicija optimalnosti je težja. Zato tukaj govorimo o sintezi in ne o optimizaciji (minimalno število vrat, literalov, minimalna zakasnitev. VLSI tehnologija porabi več prostora za povezave kot logiko - zato so literali, ki so soodvisni od povezav, zelo pomembni.)

• Dinamično obnašanje večnivojske logike je pomembno, saj lahko vodi do hazardov, ki so nezaželeni. Postopek sinteze mora upoštevati vzroke, ki vodijo do hazardov.

• Sinteza večnivojske logike ima 2 koraka:

1. Prvi je tehnološko neodvisen in določa skupne gradnike - faktorje s ciljem, da se reducira "fan-in" za ceno dodatnih nivojev. Uporablja osnovne

lastnosti Boolovih izrazov.

2. Drugi je tehnološko odvisen in preslika faktorsko obliko v specifično implementacijo z uporabo knjižnice dosegljivih vrat.

4.1 Faktorske oblike in operacije

• Večnivojska sinteza pretvori Boolov izraz v faktorsko obliko, ki je izraz, ki izmenjuje operatorje AND in OR (vsota produktov vsote produktov ...), npr.:

$$F = (x_1x_2 \vee \overline{x_2}x_3)[x_3 \vee x_4(x_5 \vee x_1\overline{x_3})] \vee (x_4 \vee x_5)(x_6 \cdot x_7)$$

Tukaj nadaljna faktorizacija ni možna. Zaradi štetja literalov, zapišemo F kot niz dvonivojskih izrazov:

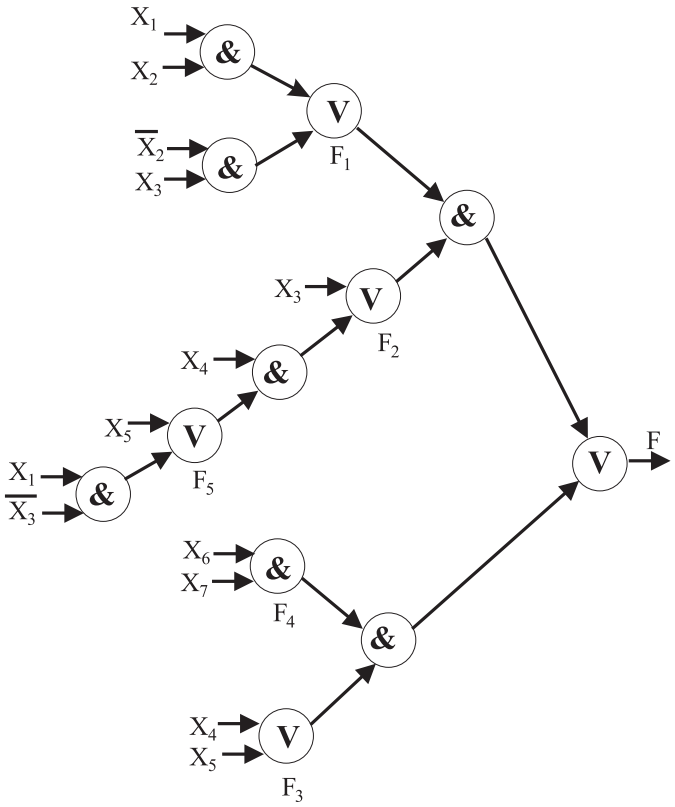
			Literali
$F = F_1 \cdot F_2$	$\vee$	$F_3 \cdot F_4$	4
F_1	$=$	$x_1x_2 \vee \overline{x_2}x_3$	4
F_2	$=$	$x_3 \vee x_4 \cdot F_5$	3
F_3	$=$	$x_4 \vee x_5$	2
F_4	$=$	$x_6 \cdot x_7$	2
F_5	$=$	$x_5 \vee x_1 \cdot \overline{x_3}$	3
			Vsota = 18

• Faktorsko obliko podajamo lahko tudi v obliki grafa oziroma drevesa (glej sliko 4.1), kjer listi predstavljajo literale, notranja vozlišča pa operacijo AND ali OR. Večinoma si operatorji AND in OR sledijo izmenično glede na sosednost v drevesu. Kot literale štejemo tudi notranja vozlišča z labelo. Funkcija F na sliki 4.1 ima 18 literalov (13 listov in 5 vozlišč z labelami).

• Osnovne operacije pri večnivojski sintezi so:

1. Dekompozicija

2. Ekstrakcija



Slika 4.1: Drevo.

3. Faktorizacija

4. Substitucija

5. Kolaps (razpad)

4.1.1 Dekompozicija

Zamenja Boolov izraz s kolekcijo novih izrazov. Npr.:

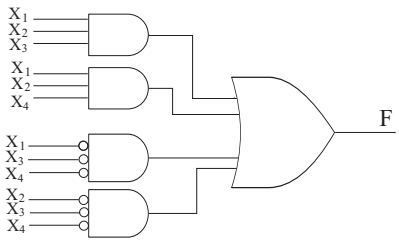
$$F = x_1x_2x_3 \vee x_1x_2x_4 \vee \overline{x_1} \cdot \overline{x_3} \cdot \overline{x_4} \vee \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4}$$

Ta izraz ima 12 literalov in zahteva 9 vrat, skupaj z operatorji negacije. Lahko pa izvedemo dekompozicijo v dve enostavnejši funkciji:

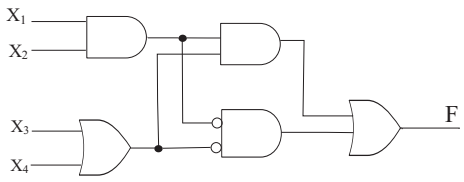
$$\begin{array}{rcll} F = F_1 \cdot F_2 & \vee & \overline{F_1} \cdot \overline{F_2} & 4 \\ F_1 & = & x_1 \cdot x_2 & 2 \\ F_2 & = & x_3 \vee x_4 & 2 \\ & & & \overline{8} \end{array}$$

Novi izraz ima skupaj 8 literalov in zahteva 7 vrat. Ker pa ni nič zastonj, se je število nivojev povečalo na 3.

Primerjava:



Slika 4.2: Dvonivojska implementacija ima 12 literalov in 9 vrat (5 operatorjev in 4 negatorje).



Slika 4.3: Tronivojska implementacija ima 8 literalov in 7 vrat (5 operatorjev in 2 negatorja).

4.1.2 Ekstrakcija

Če uporabljamo dekompozicijo nad eno funkcijo, uporabljamo ekstrakcijo nad nizom funkcij. Pri nizu funkcij išče skupne podizraze (gradnike, faktorje). Je najtežja večnivojska operacija. Npr.:

$$F = (x_1 \vee x_2)x_3x_4 \vee x_5$$

$$G = (x_1 \vee x_2)\overline{x_5}$$

$$H = x_3 x_4 x_5$$

Za ta izraz najdemo 11 literalov, implementiramo pa ga z 8 vrati (glej sliko 4.4). Z uporabo ekstrakcije (na vhodu ne zahteva funkcij v dvonivojski obliki) opazimo skupne gradnike $(x_1 \vee x_2)$ in $(x_3 \cdot x_4)$, ki nam dajo nov zapis niza funkcij:

$$F = F_1 \cdot F_2 \vee x_5$$

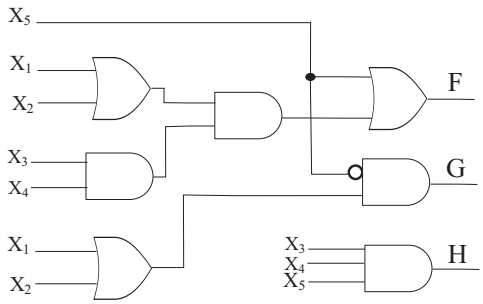
$$G = F_1 \cdot \overline{x_5}$$

$$H = F_2 \cdot x_5$$

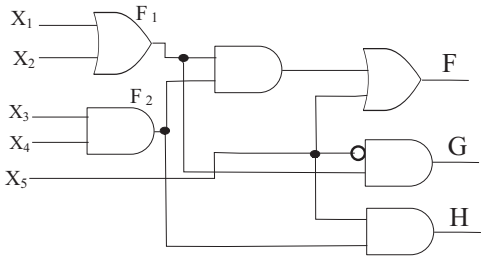
$$F_1 = x_1 \vee x_2$$

$$F_2 = x_3 \cdot x_4$$

Pri tem nizu najdemo 11 literalov, implementacija pa zahteva 7 vrat (glej sliko 4.5).



Slika 4.4: Osnovni niz: 11 literalov in 8 vrat



Slika 4.5: Po ekstrakciji: 11 literalov in 7 vrat. Funkcija H je postala 2-nivojska (večja zakasnitev).

4.1.3 Faktorizacija

Vzame izraz v dvonivojski obliki in ga izrazi kot večnivojsko funkcijo brez vpeljave labeliranih gradnikov. Uporablja se pred ekstrakcijo, da se identificira potencialne skupne gradnike. Npr.:

$$F = x_1x_3 \vee x_1x_4 \vee x_2x_3 \vee x_2x_4 \vee x_5$$

Ta funkcija ima 9 literalov in rabi 5 vrat. Po faktorizaciji dobimo:

$$F = (x_1 \vee x_2)(x_3 \vee x_4) \vee x_5$$

Da določimo število literalov, jo zapišemo kot niz dvonivojskih funkcij:

$$F = F_1F_2 \vee x_5$$
$$F_1 = x_1 \vee x_2$$
$$F_2 = x_3 \vee x_4$$

Tukaj najdemo 7 literalov in 4 vrata.

4.1.4 Substitucija

V funkcijo vnesemo nadomestno funkcijo, oziroma F izrazimo s substitutom G. Npr.:

$$F = x_1 \vee x_2x_3 \quad \rightarrow \quad F = G \cdot (x_1 \vee x_3), \quad G = x_1 \vee x_2$$

Imamo 5 literalov in 3 vrata. Sledi identifikacija skupnih izrazov:

$$F = (x_1 \vee x_2)(x_1 \vee x_3) = x_1 \vee x_2x_3$$

4.1.5 Kolaps (razpad)

To je inverzna funkcija od substitucije. Uporablja se za redukcijo števila nivojev, da se pohitri procesiranje. Npr.:

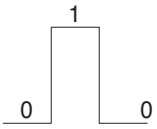
$$F = G(x_1 \vee x_3) = (x_1 \vee x_2)(x_1 \vee x_3) = x_1x_1 \vee x_1x_2 \vee x_1x_3 \vee x_2x_3 =$$
$$= x_1 \vee x_2x_3$$

Dobimo 3 literale in 2 vrata.

4.2 Hazardi in kako jih odpravimo

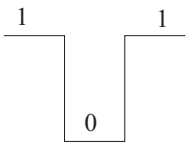
• Hazardi so posledica neenakomernih zakasnitev na različnih poteh v vezju. Obstajajo različni tipi hazardov:

1. Statični hazardi: Na Sliki 4.6 vidimo primer statičnega 0 hazarda. Na



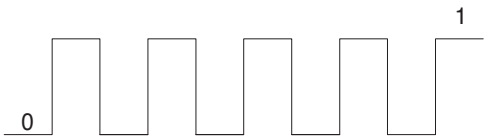
Slika 4.6: Statični 0 hazard

Sliki 4.7 pa vidimo primer statičnega 1 hazarda.

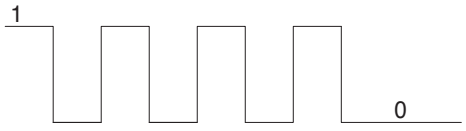


Slika 4.7: Statični 1 hazard

2. Dinamični hazardi: Na Sliki 4.8 vidimo primer dinamičnega $0 \rightarrow 1$ hazarda, na Sliki 4.9 pa primer dinamičnega $1 \rightarrow 0$ hazarda.



Slika 4.8: Dinamični $0 \rightarrow 1$ hazard



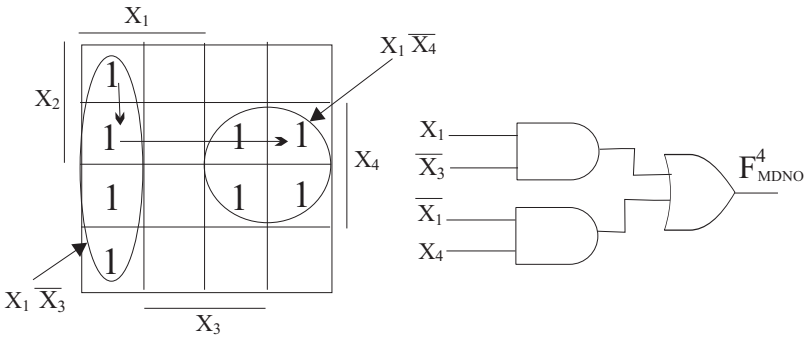
Slika 4.9: Dinamični $1 \rightarrow 0$ hazard

• Včasih se jim lahko izognemo z uporabo ure ali s povečanjem intervala med časom spremembe vhoda v vezje in časom opazovanja izhodov iz vezja. V primeru asinhronih vhodov (npr. Reset FF) pa se jim moramo izogniti s posebnim postopkom snovanja.

4.2.1 Detekcija in eliminacija statičnih hazardov v dvonivojskih vezjih

Primer: $F^4 = \vee(1, 3, 5, 7, 8, 9, 12, 13)$

Na sliki 4.10 vidimo vezje in Veitchev diagram za našo funkcijo:
$F^4_{MDNO} = x_1\overline{x_3} \vee \overline{x_1}x_4$



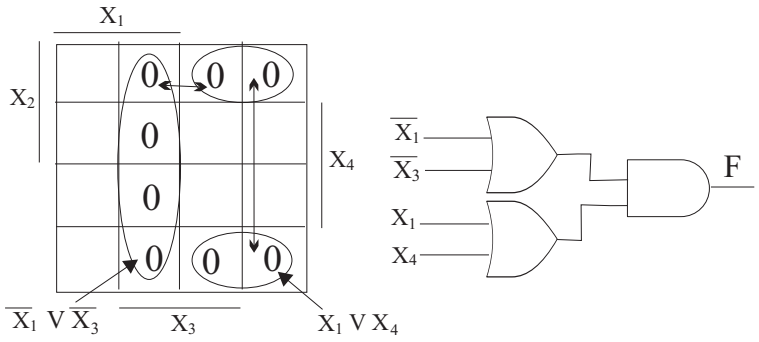
Slika 4.10: $F^4_{MDNO} = x_1\overline{x_3} \vee \overline{x_1}x_4$

• Pri spremembi na vhodu: 1100 → 1101 se F na izhodu ne spreminja (glej dogajanje v vezju), F: 1 → 1

• Pri spremembi 1101 → 0101 pa opazimo na izhodu F: 1 → 0 → 1, kar je posledica zakasnitve spremembe x_1 na vhodu $\overline{x}_1$. Vzrok je posledica spremembe vhoda iz področja enega glavnega vsebovalnika v drugega, kar lahko povzroči hazard. Z dodatkom redundančnega glavnega vsebovalnika lahko zagotovimo, da se vse enobitne vhodne spremembe dogodijo znotraj glavnih vsebovalnikov. Tedaj hazard ne nastopi več, s tem pa smo eliminirali 1-hazard.

Redundantni glavni vsebovalnik za zgornji primer je: $\overline{x_3}x_4$. Funkcija torej izgleda takole: $F^4_{DNO} = x_1\overline{x_3} \vee \overline{x_1}x_4 \vee \overline{x_3}x_4$. Sedaj sprememba: 1101 → 0101 povzroči F: 1 → 1

• Podobne razmere so pri KNO, kjer lahko nastopi 0-hazard.
Primer (glej sliko 4.11) : $F^4_{MKNO} = (\overline{x_1} \vee \overline{x_3})(x_1 \vee x_4)$



Slika 4.11: $F^4_{MKNO} = (\overline{x_1} \vee \overline{x_3})(x_1 \vee x_4)$

Pri spremembi: 0110 → 1110 nastopi F: 0 → 1 → 0 (0-hazard). Preprečimo pa ga z dodatno redundanco $(\overline{x_3} \vee x_4)$. Naša funkcija sedaj izgleda takole: $F_{KNO} = (\overline{x_1} \vee \overline{x_3})(x_1 \vee x_4)(\overline{x_3} \vee x_4)$. Rezultat je isti kot prej. $(F_{DNO} = F_{KNO})$

• Analizo 0-hazarda pa lahko napravimo tudi tako, da najprej negiramo funkcijo, ki je prosta 1-hazarda. Nato pogledamo, če negirana funkcija s svojimi glavnimi vsebovalniki pokrije vse sosednje 0 v originalni funkciji (Glej sliko 4.12).

$$F = x_1\overline{x_3} \vee \overline{x_1}x_4 \vee \overline{x_3}x_4$$

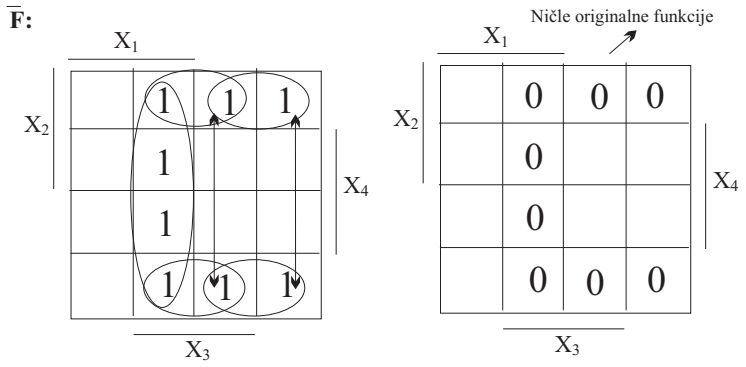
Zgornja funkcija je prosta 1-hazarda.

$$\overline{F} = (\overline{x_1} \vee x_3)(x_1 \vee \overline{x_4})(x_3 \vee \overline{x_4}) = x_1x_3 \vee x_3\overline{x_4} \vee \overline{x_1} \cdot \overline{x_4}$$

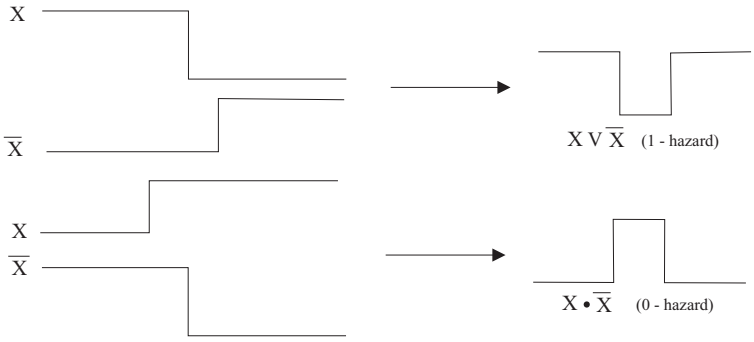
Ker glavni vsebovalniki $\overline{F}$ pokrijejo vse sosednje ničle originalne funkcije, 0-hazard ne eksistira.

4.2.2 Detekcija statičnega hazarda v večnivojskih vezjih

• Večnivojsko obliko funkcije preslikamo v dvonivojsko, ki jo imenujemo "tranzientna izhodna funkcija". Pri tem odstopamo od izrazov: $x \cdot \overline{x} = 0$ in $x \vee \overline{x} = 1$, saj le te v razmerah realnih zakasnitev ne držijo več za vse čase (primer je na sliki 4.13). Nad tranzientno izhodno funkcijo uporabimo metodo za odpravo hazardov pri dvonivojski logiki.



Slika 4.12: $\overline{F} = (\overline{x_1} \vee x_3)(x_1 \vee \overline{x_4})(x_3 \vee \overline{x_4}) = x_1x_3 \vee x_3\overline{x_4} \vee \overline{x_1} \cdot \overline{x_4}$



Slika 4.13:

Primer: $F = x_1x_2x_3 \vee (x_1 \vee x_4)(\overline{x_1} \vee \overline{x_3})$.
(7 literalov / 5 operatorjev in 2 negaciji)

Najprej z uporabo distributivnosti dobimo:

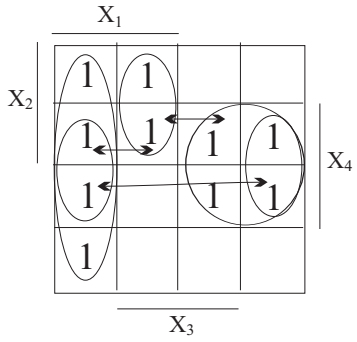
$$F = x_1x_2x_3 \vee \underbrace{x_1\overline{x_1}}_{ni\ nevaren\ za\ 1-hazard} \vee \overline{x_1}x_4 \vee x_1\overline{x_3} \vee \overline{x_3}x_4$$

Testiramo F glede na 1-hazard (vse spremembe na 1 spremenljivko).

Opazimo 1-hazard pri prehodih (glej sliko 4.14): $1111 \rightarrow 0111$ in $1111 \rightarrow 1101$.

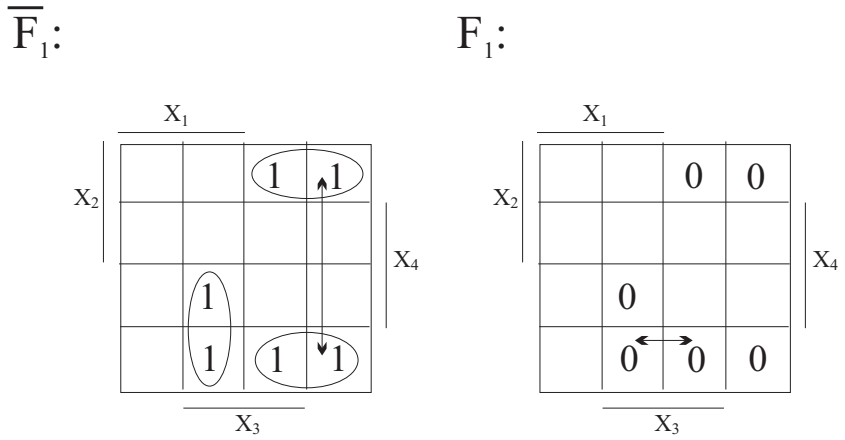
Dodati moramo 2 glavna vsebovalnika: x_1x_2 in x_2x_4. Funkcija, ki jo dobimo s tema dvema dodatnima vsebovalnikoma, je: $F_1 = x_1\overline{x_3} \vee \overline{x_1}x_4 \vee \overline{x_3}x_4 \vee x_1x_2 \vee x_2x_4$. Ustrezno vezje nima več 1-hazardov in vsebuje 10 literalov, 6 vrat ter 2 negatorja. Preverimo še 0-hazarde (s pomočjo $\overline{F_1}$).

$$\overline{F_1} = x_1\overline{x_2}x_3 \vee \overline{x_1} \cdot \overline{x_4}$$



Slika 4.14:

To vidimo na sliki 4.15. Dodati moramo še glavni vsebovalnik $\overline{x_2}x_3\overline{x_4}$ k $\overline{F_1}$.



Slika 4.15: Sosednji ničli (desni diagram) nista pokriti z istim gl. vsebovalnikom funkcije $\overline{F_1}$

$$\overline{F_2} = \overline{F_1} \vee \overline{x_2}x_3\overline{x_4}$$

$$F_2 = \overline{x_1\overline{x_2}x_3 \vee \overline{x_1} \cdot \overline{x_4} \vee \overline{x_2}x_3\overline{x_4}} = (\overline{x_1} \vee x_2 \vee \overline{x_3}) \cdot (x_1 \vee x_4) \cdot (x_2 \vee \overline{x_3} \vee x_4) = F_1$$

Ustrezno vezje je 0-hazardov prosto. Z uporabo distributivnosti (Distributivnost ne vnaša hazardov!) bi dobili končno funkcijo: $F = (\overline{x_1} \vee x_2 \vee \overline{x_3})x_4 \vee x_1(x_2 \vee \overline{x_3})$. Funkcija ima 7 literalov, za realizacijo pa porabimo 7 vrat. Ta rezultat je analogen izhodišču, le da nima hazardov.

4.2.3 Pravila za snovanje hazardov prostih vezij

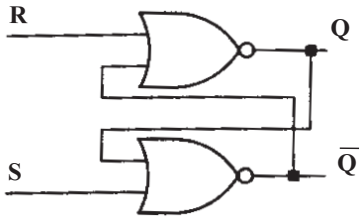
1. Velja za DNO (2 nivojska - tranzientna oblika): Vsi sosednji mintermi morajo biti pokriti z istim glavnim vsebovalnikom in noben konjunktivni izraz ne sme vsebovati $x_i \cdot \overline{x_i}$. Prvi pogoj eliminira 1-hazard, drugi pa 0-hazard.

2. Velja za KNO (2 nivojska - tranzientna oblika): Vsi sosednji makstermi morajo biti pokriti z istim glavnim vsebovalnikom (disjunkcijo) in noben disjunktivni izraz ne sme vsebovati $x_j \vee \overline{x_j}$. Prvi pogoj eliminira 0-hazard, drugi pa 1-hazard.

Poglavje 5

SEKVENČNO LOGIČNO SNOVANJE

• Osnovni pomnilni element, ki ga vidimo na sliki 5.1, dobimo s povratnima vezavama dveh NOR elementov. Tak element imenujemo tudi "RS - latch" ali RS - zatič.



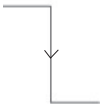
Slika 5.1: RS - zatič.

R	S	Q(t+1)
0	0	Q(t)
0	1	1
1	0	0
1	1	X

• Z dodatnim sinhronizacijskim signalom (uro) lahko časovno nadziramo spreminjanje celic v odvisnosti od vhodov in stanja. Pri uri ločimo prvo (slika 5.2)

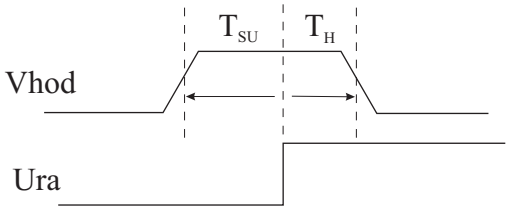
in zadnjo (slika 5.3) fronto:

Slika 5.2: prva fronta



Slika 5.3: zadnja fronta

To sta dva časovna dogodka ("clocking events"), ki povzročita spremembe v pomnilnih celicah. Pri tem je pomembno, da so vhodi, ki določajo novo stanje, stabilni v okolici fronte.



Slika 5.4: Minimalna časa v okolici fronte.

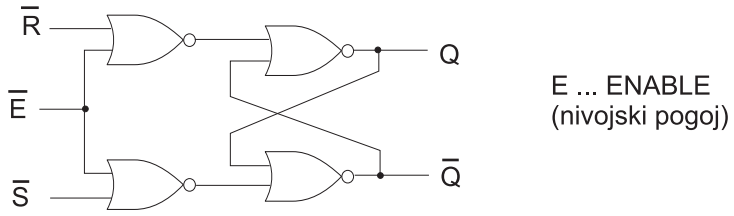
T_{SU}("set-up" čas) in T_H("Hold" čas) sta minimalna časa v okolici fronte, ko mora biti vhod stabilen.

• Osnovne pomnilne elemente delimo v dva razreda:

1. v zatiče (izhod sledi vhodu)
2. v flip-flope (FF)

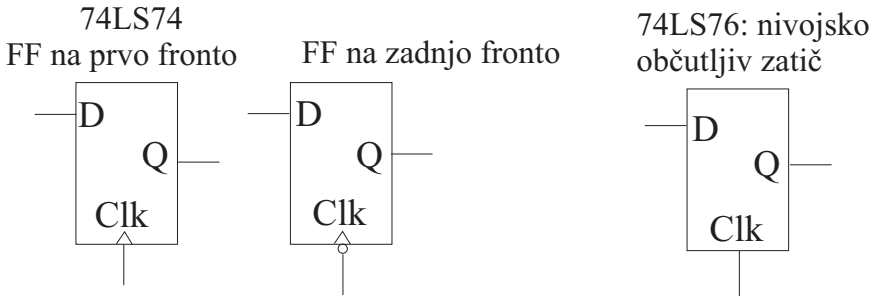
Med zatiče sodijo tudi nivojsko - občutljivi zatiči z dodatnim ("enable") vhodom, ki mu pravimo včasih tudi ura. Tako vezje vidimo na sliki 5.5.

Ko gre signal $\overline{E}$ iz 0 na 1, se na izhodu zatiča ohrani vrednost, ki jo določata vhoda znotraj okna (T_S, T_H glede na fronto). Pri FF se izhodi spremenijo



Slika 5.5: Nivojsko-občutljivi zatič.

samo v skladu z uro. Ločimo FF, prožen na 1 fronto, zadnjo fronto in MS FF. Primeri so na sliki 5.6.



Slika 5.6: Oznake FF in zatičev.

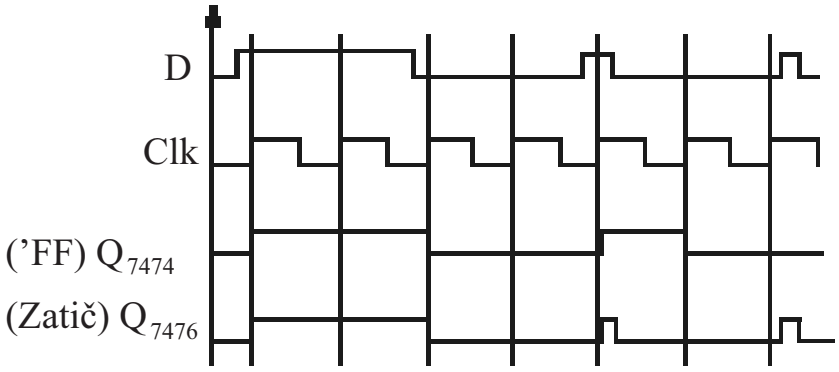
Primerjava zatičev in FF:

Tip	**Kdaj so vhodi vzorčeni**	**Kdaj so veljavni izhodi**
Navaden zatič	Vedno	Zakasnjena vhodna sprememba
Nivojsko občutljiv zatič	Visoka ura (T_{SU},T_h okoli zadnje f.)	Po zakasnjeni vhodni spremembi
FF na prvo fronto	Prva fronta (T_{SU},T_h okoli prve f.)	Po zakasnitvi glede na prvo f.
FF na zadnjo fronto	Zadnja fronta (T_{SU},T_h okoli zadnje f.)	Po zakasnitvi glede na zadnjo f.
MS FF	Zadnja fronta (T_{SU},T_h okoli zadnje f.)	Po zakasnitvi glede na zadnjo f.

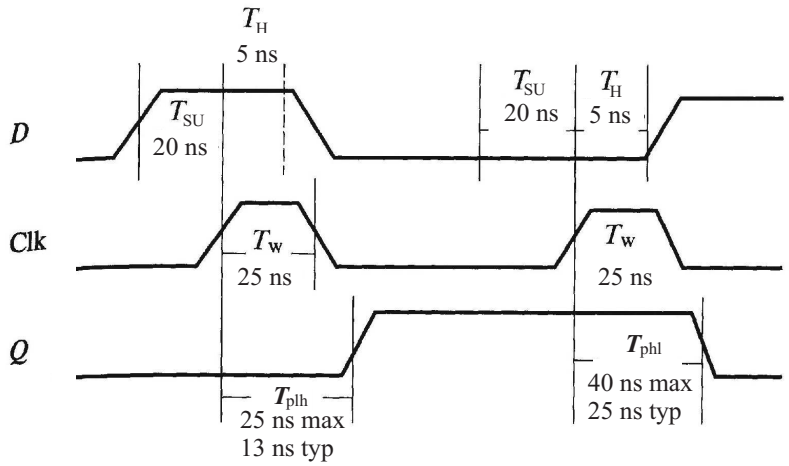
• Časovni odzivi 'FF (FF na prvo fronto) in nivojsko občutljivega zatiča so na sliki 5.7. Razlika nastopi v primerih, ko je Clk aktiven v času spremembe vhoda.

• Definicija T_{SU} in T_h za slučaj 'FF (74LS74) je vidna na sliki 5.8.

• Pri nivojsko občutljivih zatičih pa so T_{SU} in T_h definirani relativno glede na zadnjo fronto in tudi T_{phl} in T_{plh} so bolj kompleksni (dva slučaja aktivacije: od vhodne spremembe → izhodno spremembo in od spremembe ure → izhodne



Slika 5.7: Časovni odzivi.

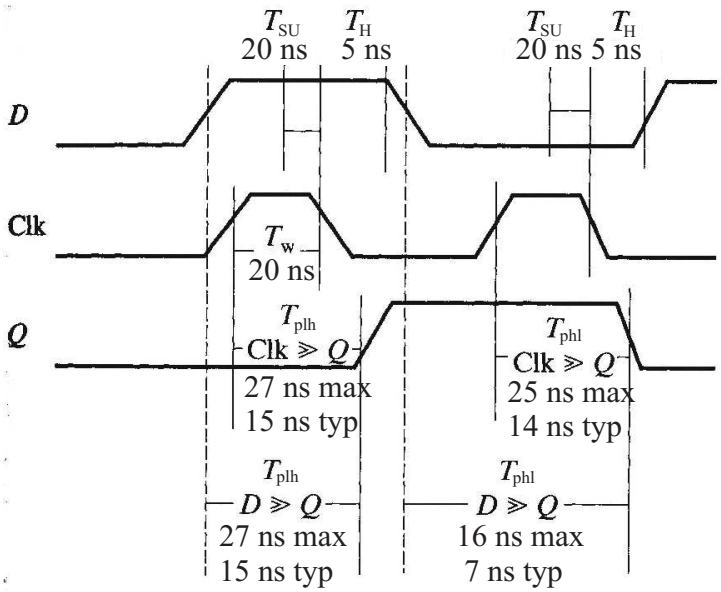


T_w - širina ure
T_{plh} - zakasnitev med vzponom ure in vzponom izhoda (propagation low to high)

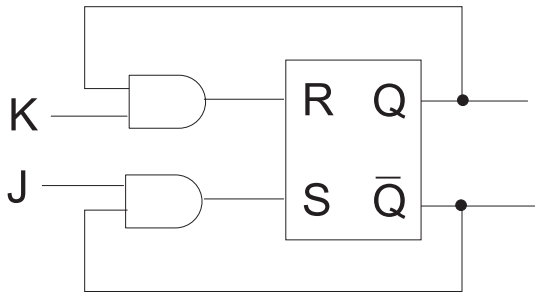
Slika 5.8:

spremembe). Seveda mora biti ura aktivna prej, preden lahko vhod vpliva na izhod. (Ker gre za nivojsko občutljive zatiče). Te razmere prikazuje slika 5.9.

• JK FF dobimo iz RS na način, ki ga prikazuje slika 5.10.



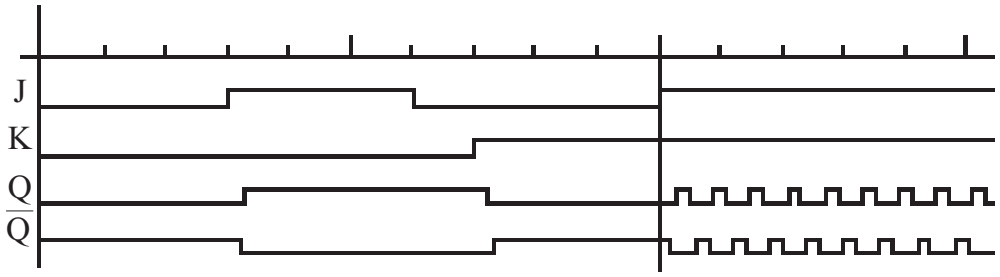
Slika 5.9:



Slika 5.10: $Q(t+1) = KQ \vee J\overline{Q}$.

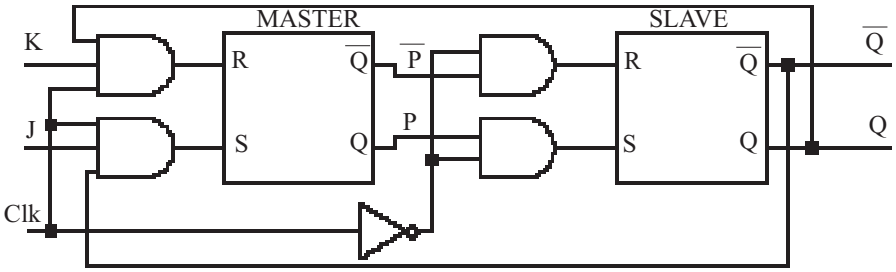
J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	$\overline{Q}$(t)

Pri J=K=1 izhod oscilira ("toggle") tako kot kaže slika 5.11.



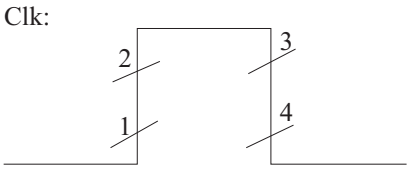
Slika 5.11: Osciliranje izhoda

• Rešitev zgornjega problema predstavlja vezje MS FF na sliki 5.12.



Slika 5.12: MS FF

Clk signal deluje pri tej rešitvi tako kot kaže slika 5.13. Pri tem upoštevamo naslednje:

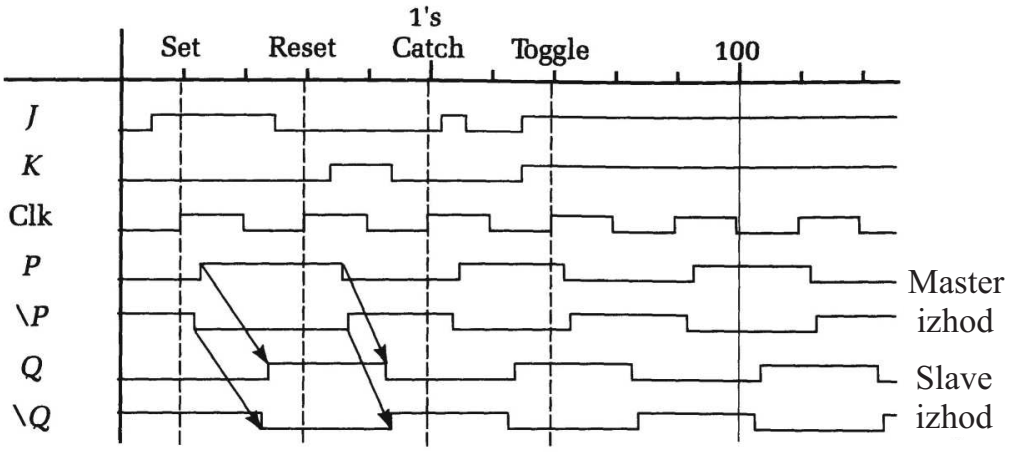


Slika 5.13: Clk signal

1. M/S - prekine se vez med M in S
2. X → M - poveže vhod z M
3. X/M - odklopi vhod od M

4. M → S - poveže M z S

• Pri MS FF pa nastopi problem "one's catching", tj., če ob visoki uri pride do spremembe na vhodu, se ta upošteva, čeprav gre za kratek impulz (hazard). Glej sliko 5.14.



Slika 5.14: Problem ujetja enice (one's catching).

MS FF so zelo občutljivi na hazarde (0 - statični hazard). Pri RS zatičih z NAND vrati je aktivni signal 0 in tedaj govorimo o "0 catch" problemu (občutljivost na 1 statične hazarde)

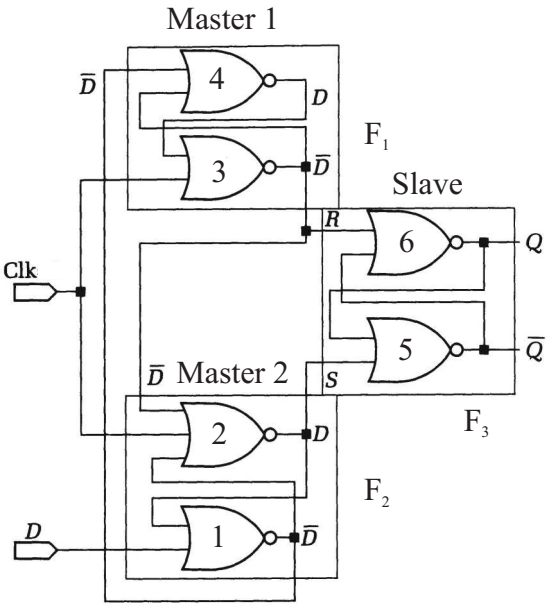
• Rešitev problemov "one's catch" oziroma "zero's catch" pa predstavljajo FF, ki delujejo na fronto. Takšni FF so bolj kompleksni od MS, saj vsebujejo 3 zatiče.

Zgled: D FF' (na zadnjo fronto), ki ga prikazuje slika 5.15:

Mirovna kombinacija: $Cl = 1 \rightarrow F_1 = F_2 = 0$, kar ohranja F_3 v prvotnem stanju.

Aktivna kombinacija: $Cl = 1 \rightarrow 0 : F_1$ in F_2 delujeta kot invertorja, ki dajeta na R in S vhode F_3 $\overline{D}$ oziroma D. ($F_1 \rightarrow \overline{D} \rightarrow R$; $F_2 \rightarrow D \rightarrow S$). To pomeni, če je D=1 → Q = 1 in če je D = 0 → Q = 0.

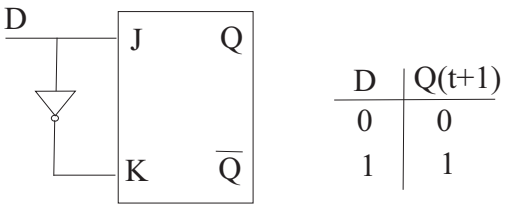
Mirovna kombinacija: Cl=0 , D → D'. Spodnji NOR v F_2 ima na vhodu D in D'($\neq$ D) zato je izhod 0. To zagotavlja, da imajo izhodi iz vrat 2,4 in 5



Slika 5.15: D FF'.

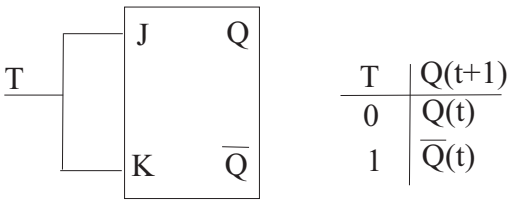
ohranjene prejšnje vrednosti.

• Iz JK FF izhajata še oba tipa FF: To sta D, ki ga vidimo na sliki 5.16 in T, ki je na sliki 5.17.



D	Q(t+1)
0	0
1	1

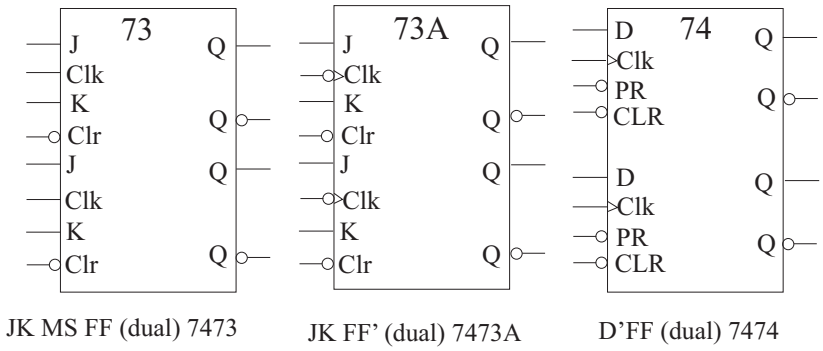
Slika 5.16: D FF



T	Q(t+1)
0	Q(t)
1	$\overline{Q}$(t)

Slika 5.17: T FF

• TTL katalog vsebuje številne RS zatiče ter JK in D FF. Najbolj popularni so na sliki 5.18.



Slika 5.18:

Običajno se RS zatiče formira z dvema NOR operatorjema (7402). Štirje RS zatiči pa so v komponenti 74279.

5.1 Metodologija povezovanja ("TIMING")

• Gre za pravila pri povezovanju komponent in urinih signalov, ki zagotavljajo pravilno delovanje sistema. V tem poglavju bomo obravnavali snovanje sinhronskih sistemov.

• Osnovno pravilo je, da se vhodi ne smejo spreminjati med spreminjanjem notranjih stanj (stanj pomnilnih celic). Znotraj T_{SU} in T_h morajo biti vhodi stabilni, izhodi pa se lahko spreminjajo kot odgovor na urine spremembe in vhodne spremembe. Vse to vpliva na kompleksne časovne metodologije oziroma pravila za povezovanje.

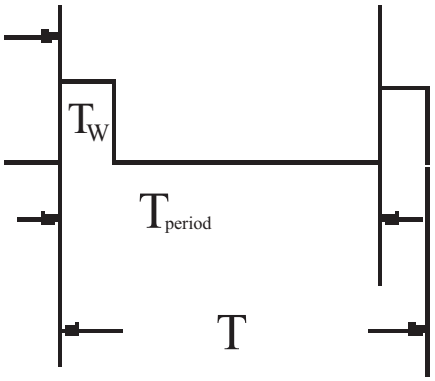
• Za pravilno delovanje sinhronskih sistemov morata biti izpolnjena dva pogoja:

1. Na vhode FF morajo prihajati pravilne vrednosti ob upoštevanju urinih dogodkov (fronta, nivo)

2. FF lahko zamenja svoje stanje samo enkrat v enem časovnem dogodku

Prvi pogoj tudi pomeni, da se vhodi vseh FF upoštevajo istočasno. To lahko predstavlja problem, npr. če vsi FF niso enako hitri, oziroma če so dolžine poti urinega signala zelo različne.

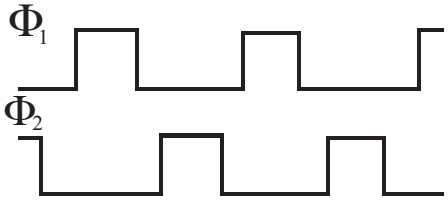
• Pri sekvenčnih vezjih z nivojsko občutljivimi zatiči lahko pride do večkratne spremembe stanja na urin dogodek. Tega problema ni pri FF na fronto, pri zatičih pa ga rešujemo s posebnimi urinimi signali s kratko dolžino trajanja impulza ("Narrow width clocking") (glej sliko 5.19) in naslednji pogoji:
$T_w << T_p$,



Slika 5.19:

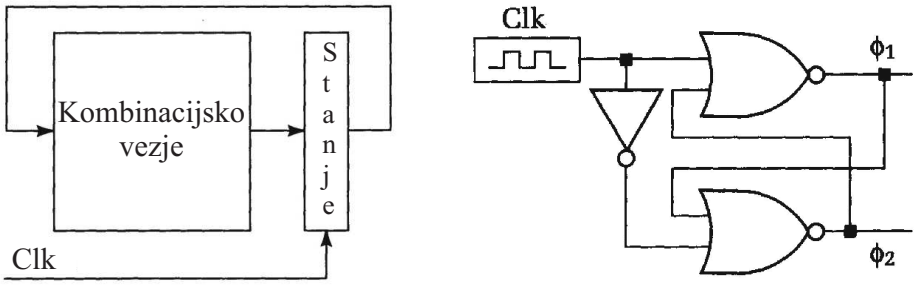
$T_w <$ najhitrejša pot v kombinacijskem delu sekvenčnega vezja plus zakasnitev zatiča,
$T_p >$ najdaljše poti v kombinacijskem vezju,
T > najdaljša zakasnitev plus T_{SU} zatiča.

• Alternativo ozkim urinim impulzom predstavlja večfazna ura ("multiphase clocking") brez prekrivanja impulzov posameznih faz. Primer 2-fazne ure brez prekrivanja je na sliki 5.20.



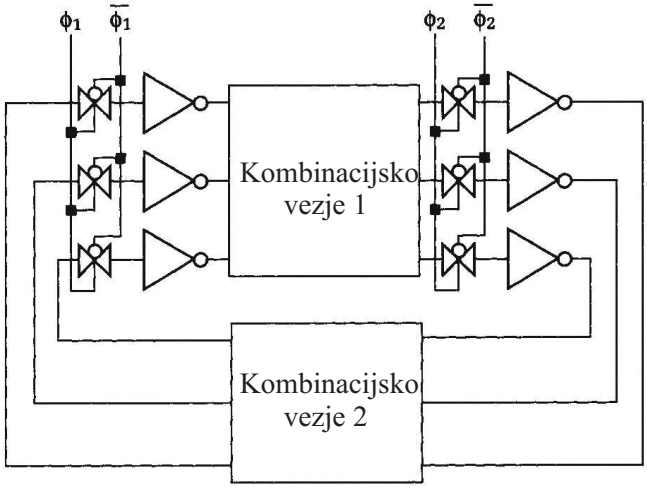
Slika 5.20: 2 fazna ura brez prekrivanja

S fazama kontroliramo povratne poti v sekvenčnem vezju, s čimer preprečujemo da bi v eni periodi ure signal več kot enkrat potoval skozi kombinacijsko vezje. Tako rešitev vidimo na splošnem sekvenčnem vezju na sliki 5.21. Če pa želimo



Slika 5.21:

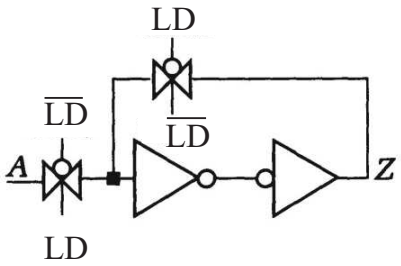
uporabo dvofazne ure, uporabimo vezje na sliki 5.22.



Slika 5.22:

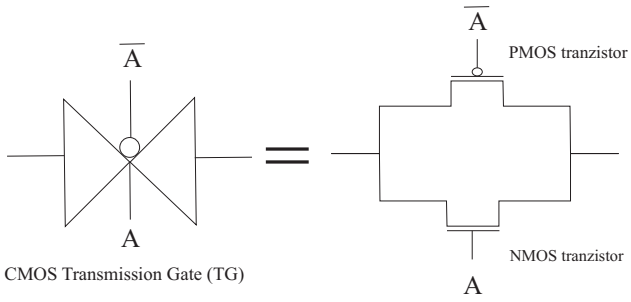
Periodi Φ_1 in Φ_2 morata biti večji od najdaljše zakasnitve v kombinacijskem vezju.

Pomnilni element z "load" in "Refresh" potmi vidimo na sliki 5.23. Ko je LD = H, A vstopa v element, povratna zanka je odprta. Ko je LD = L, se vhod (A) razklene in povratna zanka ohranja ("Refresh") vpisano vrednost.



Slika 5.23:

Razlaga vezja na sliki 5.24. Ko je A = 1, CMOS TG prevaja logično 0 ali 1 enako dobro. Pri $\bar{A} = 1$ pa je pot zaradi obeh transistorjev zaprta.



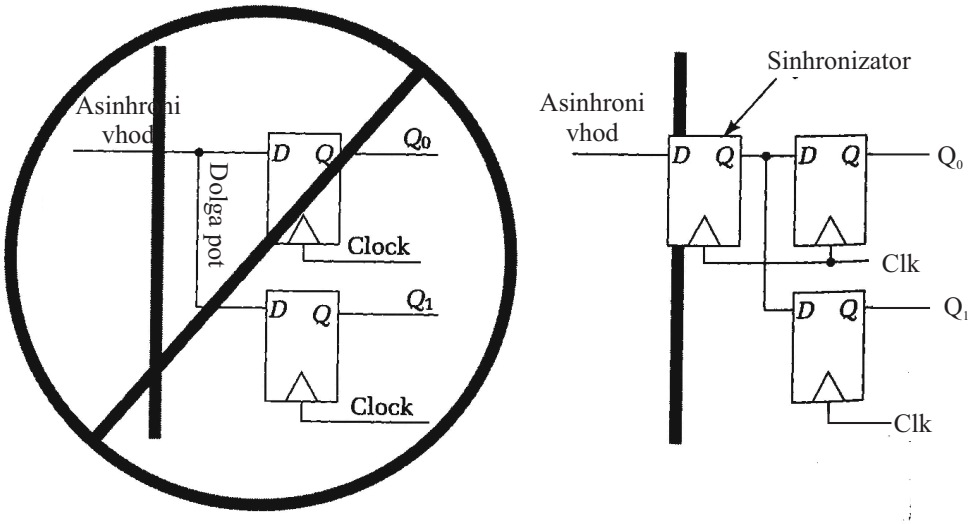
Slika 5.24:

• Pri frontnih FF je dovolj eno-fazna ura. Priporočljivo je uporabljati vse FF z istim sistemom sinhronizacije (ali 'FF ali FF')

• Probleme zakasnitve urinega signala (zaradi dolgih povezav) rešujemo tako, da povezujemo uro v obratni smeri podatkovnega toka. Če to no mogoče, je potrebno komponente, ki komunicirajo med seboj, povezati čim bliže skupaj, tako, da so urine poti kratke.

• Problem asinhronskih vhodov nastopi, če takšne vhode vodimo na več mest, ki jih sinhronizira ura. V primeru različno dolgih poti se njihov vpliv različno upošteva. Rešitev zahteva najprej "sinhronizacijo" asinhronega vhoda, nato pa distribucijo na različna mesta v vezju. To imamo prikazano na sliki 5.25.

Ta rešitev ima "napako", asinhroni vhodi ne izpolnjujejo vedno pogoja T_{SU} oziroma T_h okoli urinega dogodka, kar pomeni, da stanje sinhronizatorja ni vedno pravilno. Verjetnost napake zmanjšujemo ali s podaljšanjem urine peri-

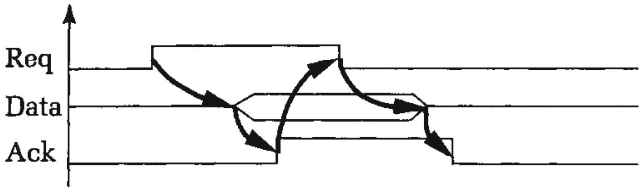


Slika 5.25:

ode ali z uvedbo dveh serijskih sinhronizatorjev. Tretji pristop k rešitvi problema asinhronih vhodov oziroma zakasnitev urinih signalov pa predstavlja tako imenovani "request/acknowledgment" izmanjava signalov, oziroma signalizacija s protokoli (Tudi komunikacija Master-Slave, Client-Server).

A: PROTOKOLI S 4 CIKLI ("FOUR - CYCLE HANDSHAKING")

Prikazuje ga časovni diagram na sliki 5.26.



Slika 5.26: "Four-cycle handshaking" protokol

Na začetku sta oba signala, REQ in ACK enaka 0:

1. faza: REQ: 0 → 1 (zahteva po podatku) MASTER → SLAVE

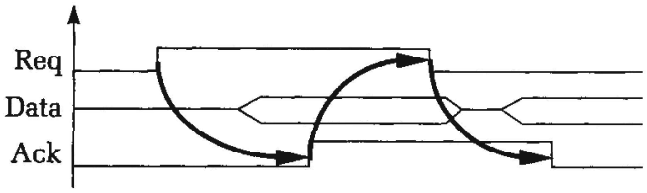
2. faza: Priprava podatkov, ACK: 0 → 1 (vračanje podatkov s potrditvijo) SLAVE → MASTER

3. faza: Čitanje podatkov, REQ: 1 → 0 (branje podatkov in umik zahteve) MASTER → SLAVE

4. faza: Umik podatkov, ACK: 1 → 0 (umik podatkov in potrditve) SLAVE → MASTER

B: PROTOKOL Z 2 CIKLOMA ("NON-RETURN TO ZERO SIGNALING")

Prikazuje ga časovni diagram na sliki 5.27.



Slika 5.27: "Non-return to zero signaling" protokol

Na začetku sta oba signala, REQ in ACK, enaka 0:

1. faza: $\overline{REQ}$: Master negira REQ in s tem da zahtevo po podatku Slave-u

2. faza: $\overline{ACK}$: Slave zazna zahtevo, pripravi podatke in odgovori z negiranjem ACK

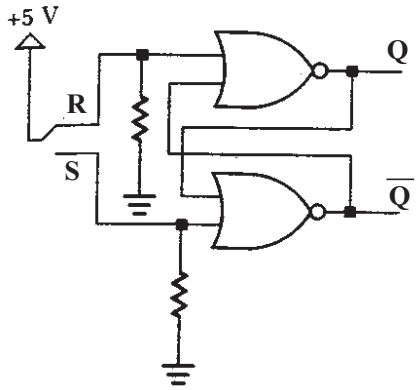
1. faza: $\overline{REQ}$: Master da novo zahtevo z negiranjem REQ potem, ko je zaznal odgovor Slave-a

2. faza: $\overline{ACK}$: Slave ukine stare podatke in na osnovi zahteve pripravi nove podatke z negiranjem ACK

5.1.1 Nekaj praktičnih rešitev

A: Preprečitev odskakovanja kontaktnika

Realni kontaktniki povzročajo odskakovanje, kar lahko preprečimo z vezavo, ki je prikazana na sliki 5.28.



Slika 5.28: Preprečitev odskakovanja kontaktnika

Stikalo v položaju R povzroči Q = 0, v položaju S pa Q = 1. Na prehodu se prejšnje stanje ohranja, kar velja tudi za slučaj odskakovanja.

B: Formiranje ure s komponento 555 ("Timer")

S komponento 555 in izborom ustreznih vrednosti za R_a, R_b in C_1 lahko realiziramo poljubno urino sekvenco. Vezje prikazuje slika 5.29.

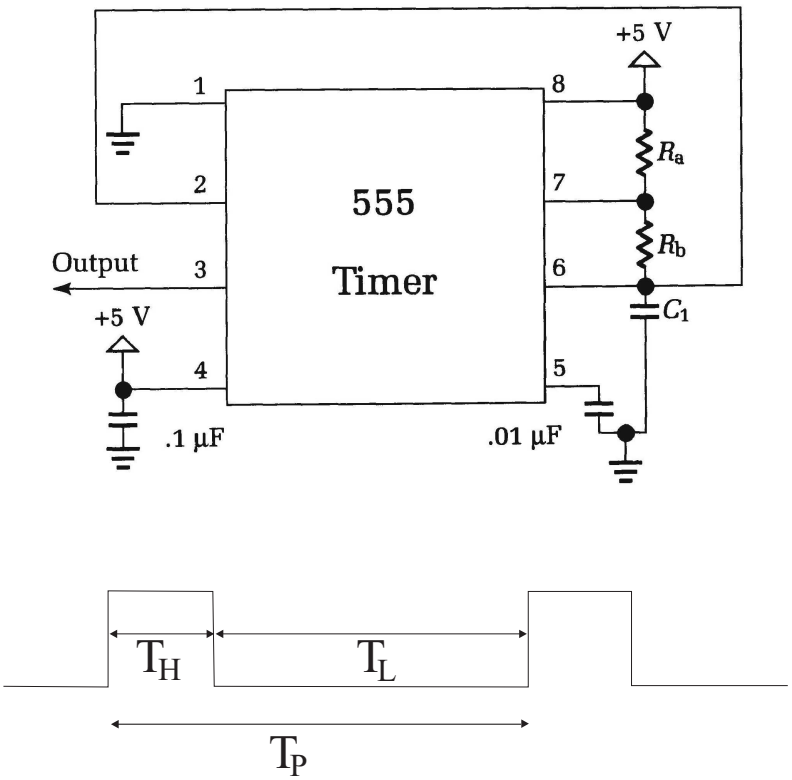
$$T_H = 0,7 \cdot (R_a + R_b) \cdot C_1$$

$$T_L = 0,7 \cdot (R_b) \cdot C_1$$

$$T_p = T_H + T_L = 0,7(R_a + 2R_b) \cdot C_1$$

$$f = \frac{1}{T_p} = \frac{DC}{0,7 \cdot (R_a + R_b) \cdot C_1}$$

DUTY CYCLE: $DC = \frac{T_H}{T_H + T_L} = \frac{R_a + R_b}{R_a + 2R_b}$



Slika 5.29: Vezje timerja 555 in njegov graf.

$R[\Omega]$, C[Faradi], R · C[Farad · Ω = sekunda]

Primer: $T_p = 500\mu$s , DC = 75%, f = $\frac{1}{T_p} = 2kHz$. Določite R_a, R_b, C_1

$$DC = 0,75 = \frac{R_a + R_b}{R_a + 2R_b} \rightarrow R_a = 2R_b$$

Izberemo: $R_a = 5k\Omega$, $R_b = 2,5k\Omega$. Pri DC=75% $\rightarrow$ $T_H = 375\mu$s, T_L=125μs

Iz enačbe: $T_L = 0{,}7 \cdot R_b \cdot C_1$ sledi:

$$125 \cdot 10^{-6} = 0,7 \cdot 2,5 \cdot 10^3 \cdot C_1$$

$$C_1 = 0,0714 \cdot 10^{-6}F = 0,0714\mu F$$

5.2 Osnovne sekvenčne komponente (registri, števci, pomnilniki)

V tem poglavju si bomo ogledali:

• Konstruiranje različnih registrov in števcev

• Postopek snovanja števcev

• Kataloški pregled dosegljivih števcev

• RAM pomnilniki: opis funkcijskih in časovnih značilnosti komponent z veliko pomnilnih elementov

5.2.1 Registri

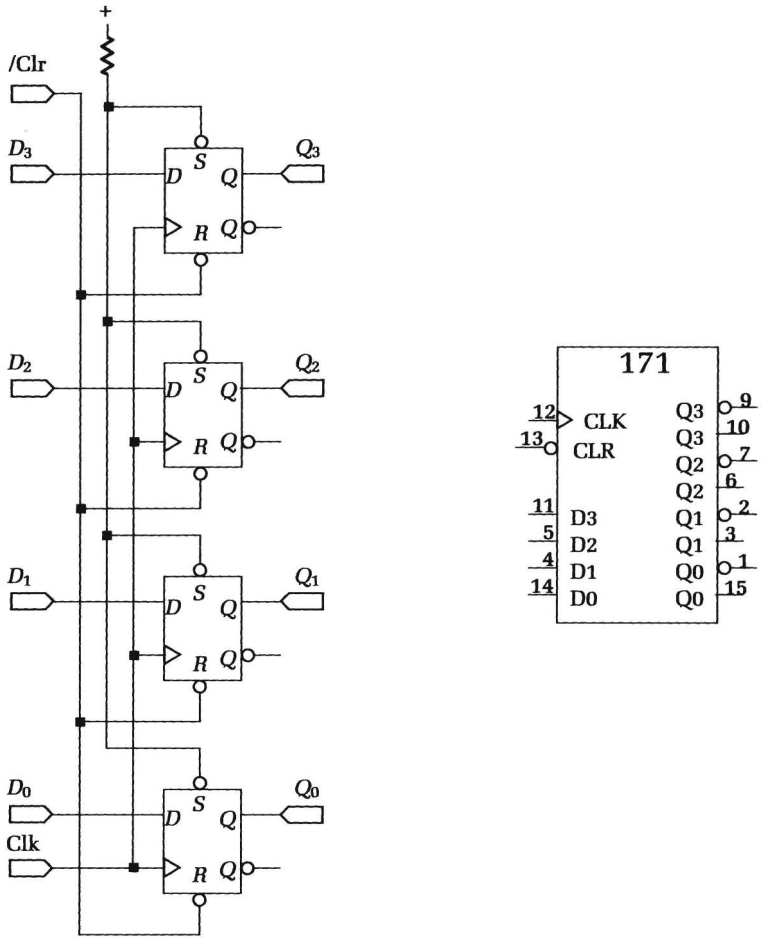
Register je komponenta več pomnilnih elementov, ki jih čitamo oziroma vpisujemo skupaj. Tipičen predstavnik je TTL 74171 (4 DFF s CLEAR), ki ga vidimo na sliki 5.30.

Komponenta 74377 ima 8 D 'FF (na prvo fronto) z $\overline{E}$ vhodom (Enable - active Low): če je $\overline{E} = 0$, potem se vpišejo novi vhodi ob 1. fronti v D FF, sicer ($\overline{E}$=1) se ohranja staro stanje.

Komponenta TTL 74374 ima 8 D 'FF z vhodom $\overline{OE}$ (Output Enable - active low): če je $\overline{OE}$=0, je vsebina registra vidna na izhodu, sicer ($\overline{OE}$=1) so izhodi odprti (v tretjem stanju - visoka impedanca).

• Več registrov tvori tako imenovano registrsko datoteko ("register file"). Vsak register tvori besedo in ima svoj indeks ali adreso. Čita in vpisuje se v posamezen register. Primer je TTL 74670 (slika 5.31): 4 X 4 registrska datoteka s 3 stanjskimi izhodi in vhodi R_B, R_A, W_B, W_A (2 bitno kodiranje za READ oziroma WRITE) ter $\overline{RE}$ (Read enable) in $\overline{WE}$. Možno je hkrati pisati in brati (različna registra).

• Razširjavo registrske datoteke predstavlja RAM (Random Access Memory). Majhen RAM je npr. 256 X 4, zelo velik pa RAM z več deset milijoni pomnilniškimi celicami. Enako težko je izvesti majhen, a hiter RAM, kot velik, a

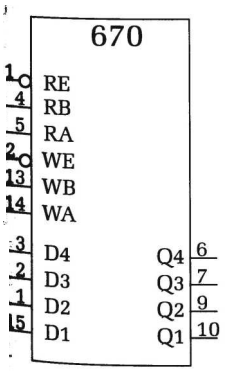


Slika 5.30: 74171

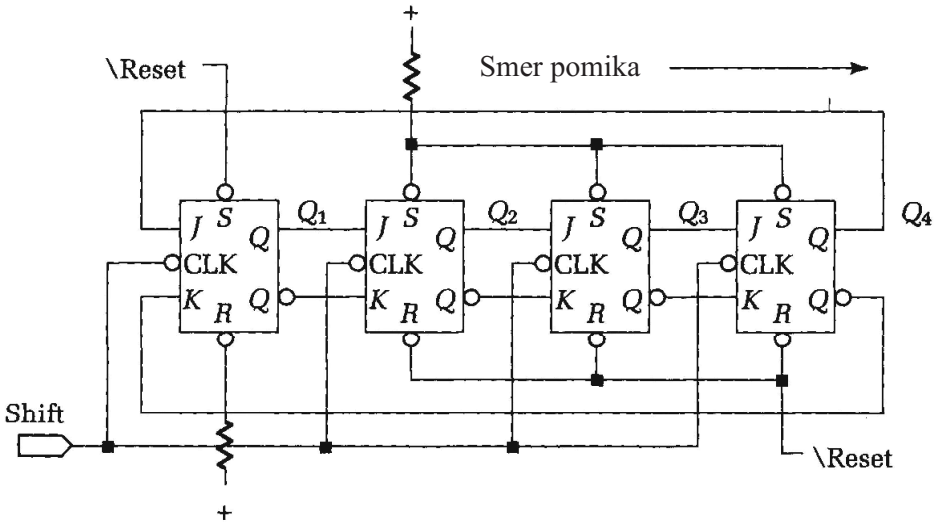
počasen pomnilnik. Več o RAM-ih v poglavju 4.2.3,

Pomikalni ("shift") registri

Omogočajo shranjevanje in ciklično pomikanje vsebine. Primer shift registra (pomik v desno) prikazuje slika 5.32.



Slika 5.31: 74670.



Slika 5.32: Primer pomikalnega registra.

začetno stanje:	1000
1. shift:	0100
2. shift:	0010
⋮	⋮

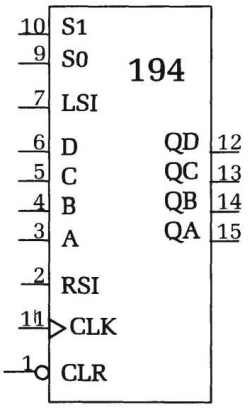
Celice so MS, zato so spremembe na izhodu ob zadnji fronti (SHIFT: $1 \rightarrow 0$)

• Glede vpisa ločujemo serijski ali paralelen vpis. Serijski je enostavnejši, saj zahteva vpis v levi FF, paralelen pa v vse FF.

• Tudi pri branju ločujemo serijski izhod (na desnem FF) ali paralelni (vsi FF so vidni hkrati)

Primer: 74194 - 4 bitni dvosmerni univerzalni shift register (glej sliko 5.33). Deluje v 4 načinih, ki jih določata kontrolna vhoda S_1 in S_0.

S_1	S_0	opis
0	0	- zadrži
0	1	- pomik v desno
1	0	- pomik v levo
1	1	- vzporedni vpis

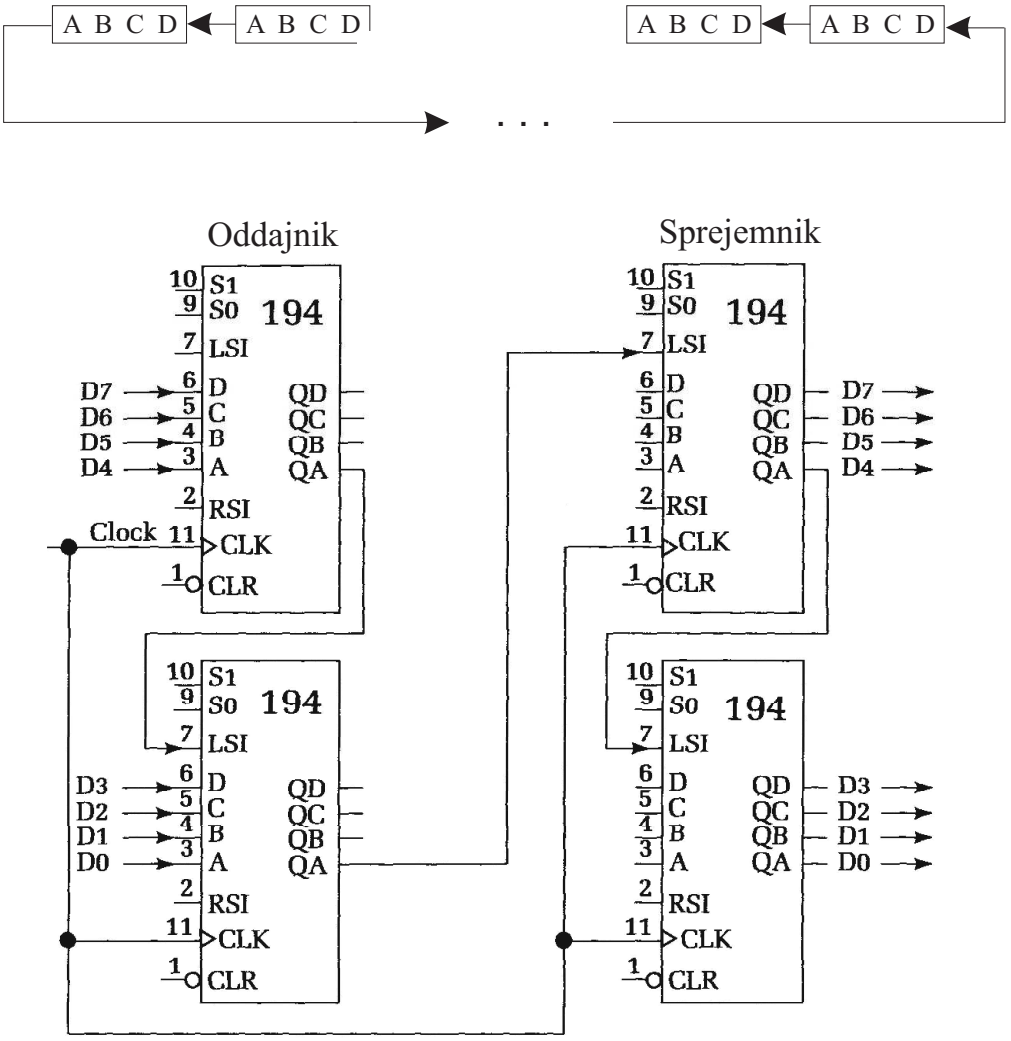


Slika 5.33: 74194.

CLR	S_1	S_0	Clk	LSI	RSI	Q_A	Q_B	Q_C	Q_D	
L	X	X	X	X	X	L	L	L	L	
H	H	H	↑	X	X	A	B	C	D	vzporedni vpis
H	L	H	↑	X	H	H	Q_A	Q_B	Q_C	pomik v desno
H	L	H	↑	X	L	L	Q_A	Q_B	Q_C	pomik v desno
H	H	L	↑	H	X	Q_B	Q_C	Q_D	H	pomik v levo
H	H	L	↑	L	X	Q_B	Q_C	Q_D	L	pomik v levo
H	L	L	X	X	X	Q_A	Q_B	Q_C	Q_D	zadrži

Legenda: LSI - Left Serial Input; RSI - Right Serial Input

• S 4 elementi 194 lahko realiziramo 8 bitni paralelno/serijski prenosni sistem (2 na oddajni in 2 na sprejemni strani). Tak sistem vidimo na Sliki 5.34 (blok shema in realizacija).



Slika 5.34: 8 bitni paralelno/serijski prenosni sistem.

Faze: LOAD na oddajni strani (11)

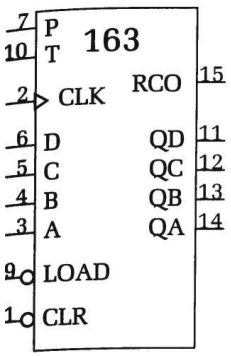
SHIFT-LEFT na oddajni in sprejemni strani (za 8 ciklov ure) (10)

HOLD na sprejemni strani (00)

5.2.2 Števci

• So sekvenčna vezja, katerih izhodi so kar notranja stanja. Stanja se periodično spreminjajo (navzgor, navzdol, krožno (pomikalni register)).

• Tipičen primer iz TTL kataloga je 74163 - sinhronski 4-bitni števec, ki ga vidimo na sliki 5.35. Ima 1 kontrolni izhod, 4 kontrolne vhode, 4 podatkovne



Slika 5.35: 74163.

vhode in 4 podatkovne izhode. Legenda k vezju:

P,T: Count Enable (Če sta 1), sicer Disable

$\overline{Load}$: omogoča vpis (je sinhronski - vezan na uro)

$\overline{Clr}$: omogoča reset (je sinhronski - vezan na uro)

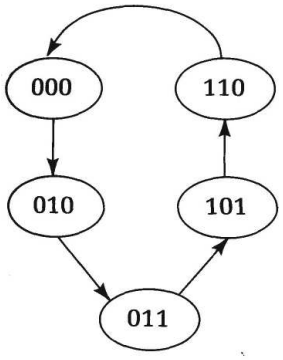
RCO: ("Ripple Carry Output"). Ko izhod doseže 15 (1111), se setira RCO (0 $\rightarrow$ 1)

• Snovanje števca ima vse elemente klasične sinteze končnih avtomatov. Pri števcu z poljubno frekvenco so koraki snovanja naslednji:

1. Na osnovi specifikacije nariši diagram prehajanja stanj
2. Nariši tabelo prehajanja stanj na osnovi diagrama prehajanja stanj
3. Vsakemu bitu v stanju ustreza svoj FF. Za vsak bit določimo krmilno enačbo oziroma funkcijo (Veitch-ev diagram)

4. Izberemo tip FF in določimo krmilne vhode za vse celice.

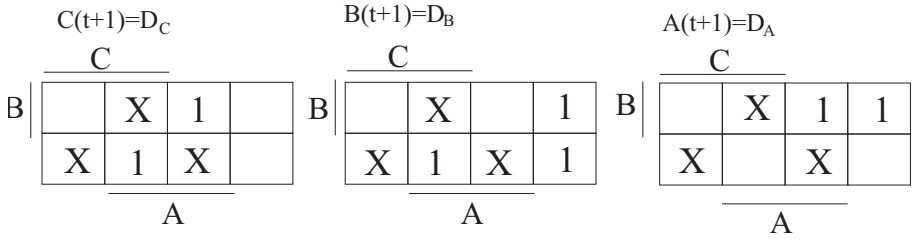
Primer: Realiziraj 3 bitni števec (glej sliko 5.36) s sekvenco: 000 → 010 → 011 → 101 → 110 → 000.



Slika 5.36: 3-bitni števec.

C	B	A	C(t+1)	B(t+1)	A(t+1)	T_C	T_B	T_A
0	0	0	0	1	0	0	1	0
0	0	1	X	X	X	X	X	X
0	1	0	0	1	1	0	0	1
0	1	1	1	0	1	1	1	0
1	0	0	X	X	X	X	X	X
1	0	1	1	1	0	0	1	1
1	1	0	0	0	0	1	1	0
1	1	1	X	X	X	X	X	X

3. točko reševanja predstavlja slika 5.37.

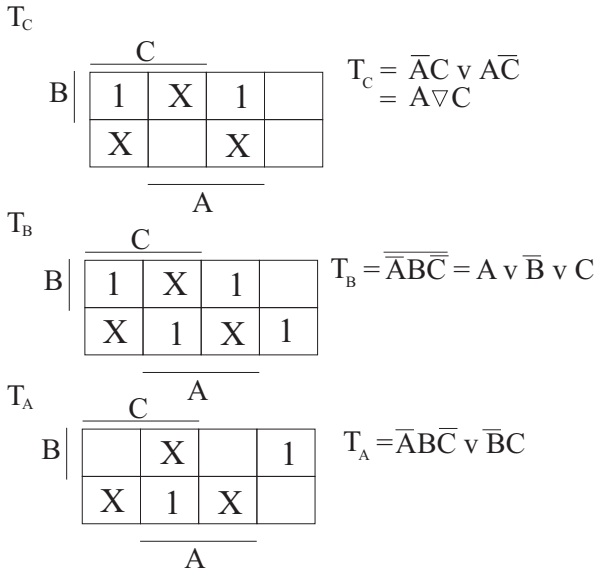


Slika 5.37:

Točka 4.) Najenostavnejše je, da izberemo D FF. Tedaj nam minimizacija funkcij iz 3. točke že daje krmilne funkcije. Sicer pa moramo dodati v tabelo vhode v izbrane celice in njihove funkcije minimizirati.

Izberemo D FF: $D_C = A$, $D_B = AC \vee \overline{A} \cdot \overline{C}$ in $D_A = B\overline{C}$

Če pa izberemo T FF, dobimo naslednje (slika 5.38).



Slika 5.38:

Vezje naše rešitve pa vidimo na sliki 5.39.

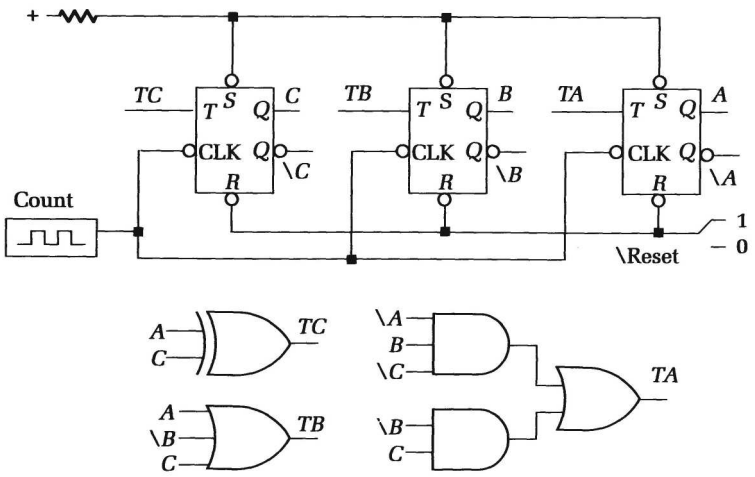
Asinhronski števci ("Ripple counters")

Ti elementi štejejo tako, da prenašajo ("rippling") odločitev o spremembi stanja iz ene celice na sosednjo.

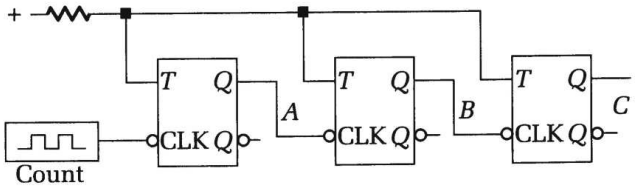
Primer 3-bitnega števca s T FF vidimo na sliki 5.40.

• To tehniko se uporablja tudi pri sestavljanju večjih števcev iz manjših. Dobljeni števci se imenujejo kaskadni števci ("cascaded counters").

Primer: 8 bitni kaskadni števec iz dveh 163 štiri bitnih sinhronskih števcev

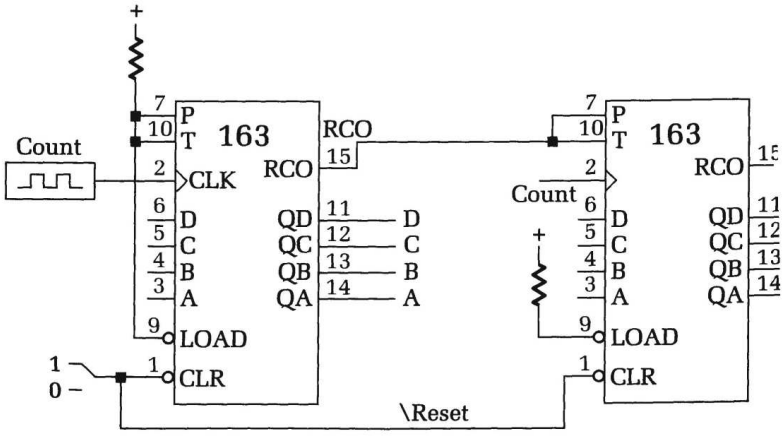


Slika 5.39: Rešitev z T FF.



Slika 5.40: Asinhronski števec s T FF.

vidimo na sliki 5.41.



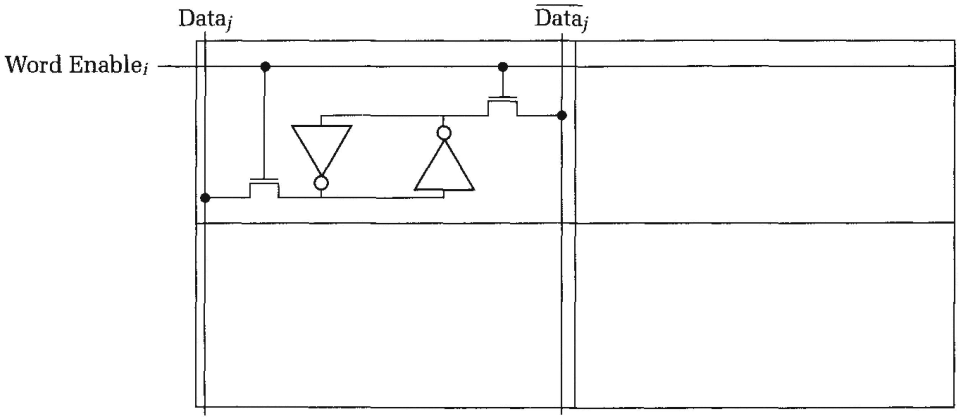
Slika 5.41: 8-bitni kaskadni števec.

5.2.3 RAM pomnilniki

• Tukaj nas bo zanimalo časovno obnašanje RAM pomnilnikov ter snovanje registrov in krmilne logike v okolici RAM-ov.

Statični RAM (SRAM)

• Osnovni pomnilni element statičnega RAM-a (slika 5.42) je element/vezje s 6 tranzistorji (Negator (CMOS) ima 2 tranzistorja). Pomnilni element real-



Slika 5.42: Osnovna celica SRAM-a.

izirata sklopljena negatorja. Tukaj ni potrebna ura ali signal za osveževanje ("refresh"), zato je hitrejši od dinamičnega RAM-a.

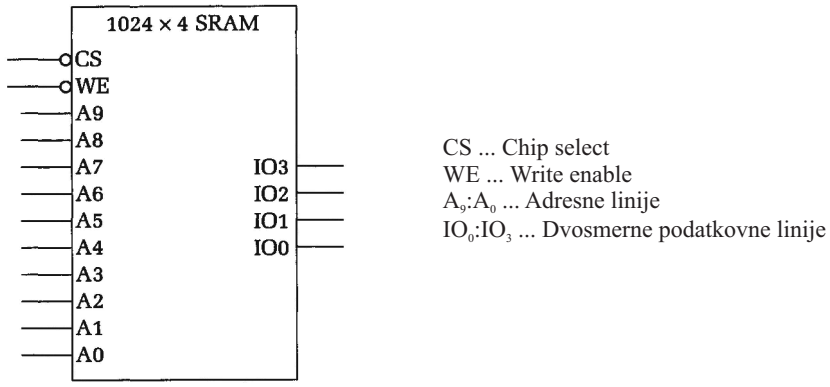
WRITE:
WE = 1, $x \rightarrow DATA_j$; $\overline{x} \rightarrow \overline{DATA}_j$, WE - "Write Enable"

READ:
WE = 1
IF $DATA_j > \overline{DATA_j} \rightarrow$ Q = 1 (to izvaja tipalni ojačevalnik)
ELSE Q = 0

• Ker je osnovne celice SRAM-a enostavno aranžirati v vrstice in stolpce in ker imajo malo tranzistorjev (6), so primerne za velike gostote. Cela vrstica si deli WE signal in cel stolpec si deli DATA signala. Običajno imajo RAM-i

1, 4 ali 8 bitov v vrstici. Širše vrstice sestavljamo iz paralelnih manjših RAM komponent.

PRIMER: SRAM 1024 X 4, ki je predstavljen na sliki 5.43.



Slika 5.43: SRAM 1024 x 4.

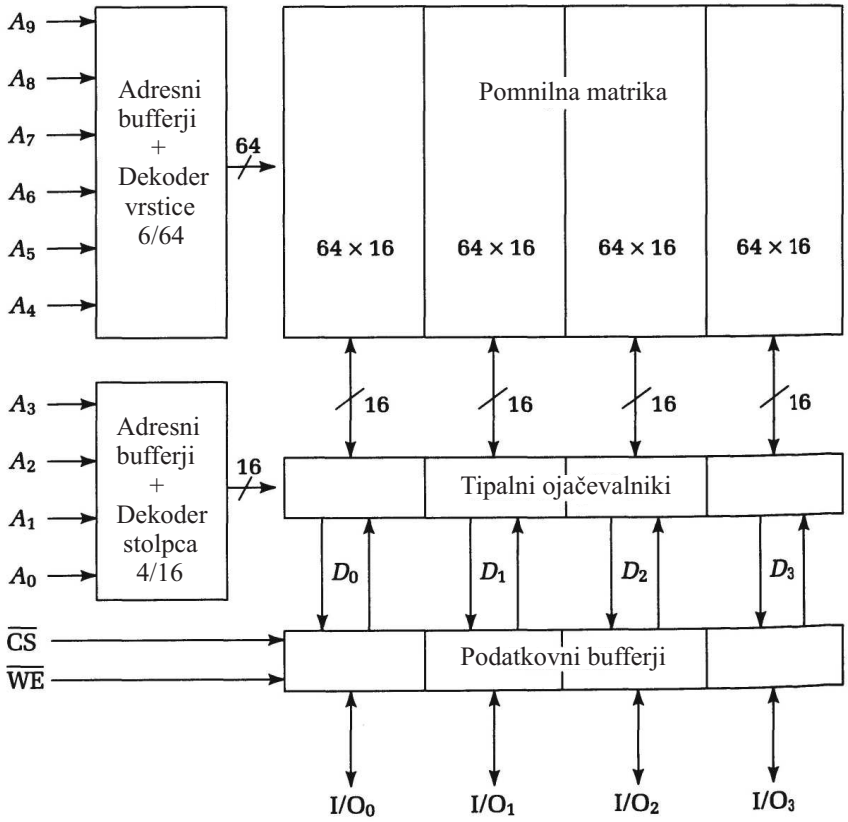
• Notranja organizacija teži po kvadratni strukturi zaradi čim krajših poti. Notranja struktura RAM-a 1024 X 4 je zato takšna kot jo je videti na sliki 5.44.

• READ Timing takšnega RAM-a vidimo na sliki 5.45.

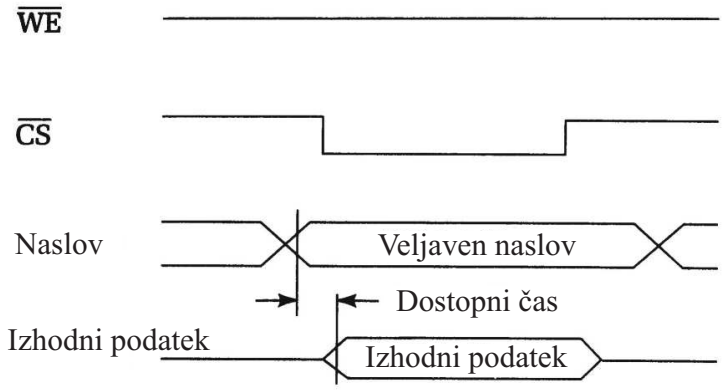
1. Veljavna adresa
2. $\overline{CS}$ $(1 \rightarrow 0)$, $\overline{WE} = 1$
3. Po dostopnem času (”access time”) so podatki na voljo na izhodu (pogoj $\overline{CS} = 0$)

• WRITE Timing takšnega RAM-a pa je prikazan na sliki 5.46:

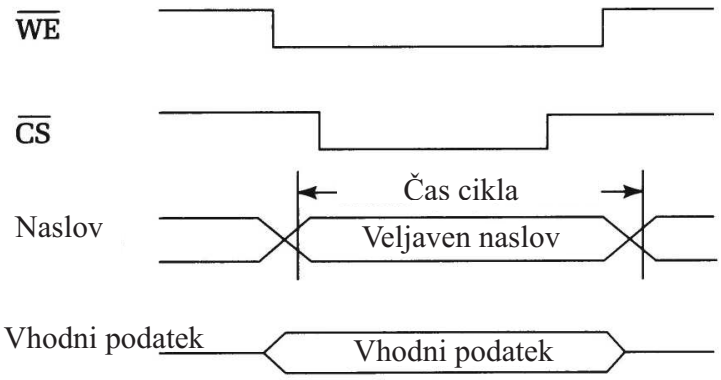
1. $\overline{WE} = 0$
2. ADDRESS in DATA IN stabilna
3. $\overline{CS} = 0$ (vpis)



Slika 5.44: Notranja organizacija SRAM-a 1024 x 4.



Slika 5.45: Read Timing 1024 x 4 RAM-a



Slika 5.46: Write Timing SRAM-a 1024 x 4.

4. $\overline{CS} = 1$
5. $\overline{WE} = 1$

Čas cikla (CYCLE TIME) je čas med dvema pomnilniškima operacijama. Velja: Čas dostopa (ACCESS TIME) $\leq$ Čas cikla.

• Statični RAM je najhitrejši in najlažji za povezovanje ("interfacing"). Največjo gostoto pa dosežemo z dinamičnim RAM-om (1 tranzistor / celico)

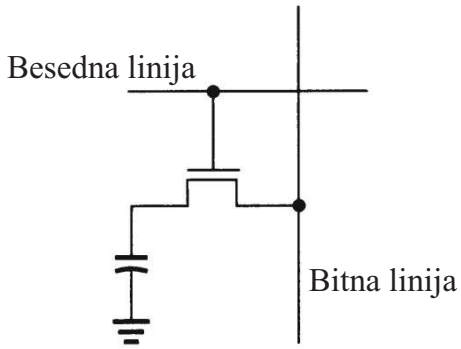
Dinamični RAM (DRAM)

Osnovna celica DRAM pomnilnika je prikazana na sliki 5.47.

Pomnilni element (celica) ima 1 tranzistor in 1 kondenzator.

WRITE: WORD LINE = 1, x $\rightarrow$ BIT LINE

READ: WORD LINE = 1
BIT LINE proti napetosti kondenzatorja določa vrednost v izbrani celici.
Pri branju vsebine kondenzatorja moramo kondenzator sprazniti preko BIT LINIJE. Zato mora zunanje vezje v DRAM shraniti vsebino in jo nato ponovno zapisati v celico.
DRAM-i izgubljajo vsebino, zato potrebujejo osveževanje ("refresh"). S fiksno



Slika 5.47: Osnovna celica pomnilnika DRAM.

periodo (reda μs) je potrebno celice brati in ponovno vpisati vsebino nazaj. Pri tem je enota vrstica. Osveževalne cikle generira zunanji pomnilni krmilnik.

Poglavje 6

SNOVANJE KONČNIH AVTOMATOV

• Ogledali si bomo metode za opisovanje obnašanja končnih avtomatov kot so: diagram prehajanja stanj, tabela prehajanja stanj(kanonska tabela), hardverski opisni jezik.
• Sledila bo predstavitev bolj formalnih opisov obnašanja končnih avtomatov.

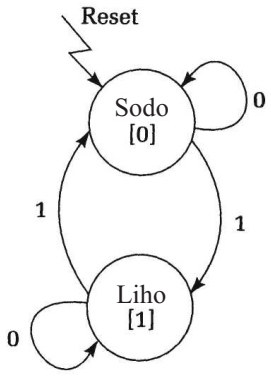
6.1 Osnovni pojmi v zvezi s končnimi avtomati

• Razložili jih bomo na primeru kontrole paritete. Kontrola prioritete mora upoštevati vso zgodovino vhodov pri določanju izhoda. Če je število enic na vhodu liho, je izhod 1, sicer 0 (liha pariteta).

• Na sliki 6.1 vidimo diagram prehajanja stanj za naš primer.

• Tabela prehajanja stanj (simbolična):

STANJE(t)	VHOD	STANJE(t+1)	IZHOD (Moorov)
SODO	0	SODO	0
SODO	1	LIHO	0
LIHO	0	LIHO	1
LIHO	1	SODO	1



Slika 6.1: Diagram prehajanja stanj za primer kontrole paritete

• Kodirna tabela prehajanja stanj (TPS):

PS	I	NS	O	
0	0	0	0	PS - present state
0	1	1	0	NS - next state
1	0	1	1	I - input
1	1	0	1	O - Output

• Funkcija prehajanja stanj in izhodna funkcija:

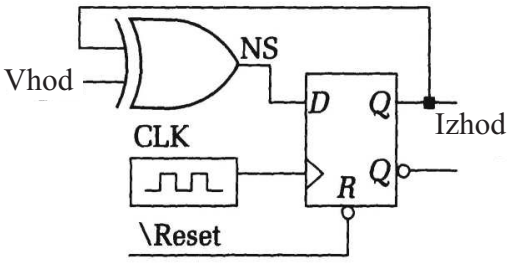
$NS = PS \bigtriangledown I$
$O = PS$

• Implementacija z D FF(vezje vidimo na sliki 6.2):

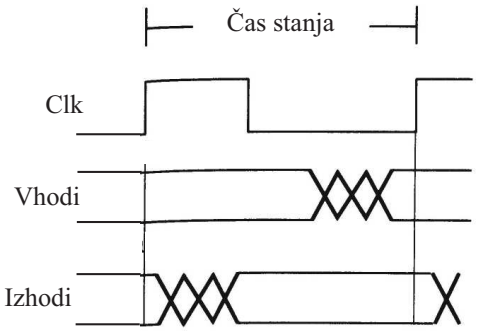
PS	I	NS	O	D
0	0	0	0	0
0	1	1	0	1
1	0	1	1	1
1	1	0	1	0

$D = NS = PS \bigtriangledown I$;
O = PS

• Časovnost v KA: Čas stanja (glej sliko 6.3) je čas med dvema urinima dogodkoma (frontama). Kot odziv na uro (1. fronto) se spremenita stanje in izhod. Vhodi morajo biti stabilni pred uro. Ob fronti se "skanirata" vhod in



Slika 6.2: Implementacija z D FF

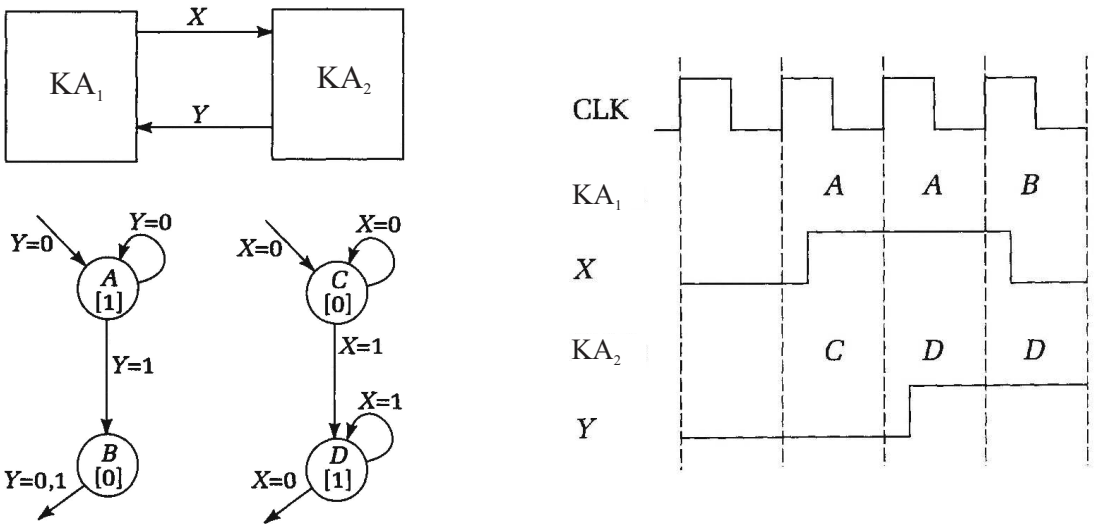


Slika 6.3: Čas stanja.

stanje, temu sledi sprememba stanja, nato še nov izhod (Moore).

- Primer veljavnosti izhoda prikazuje slika 6.4.
- Osnovni koraki snovanja splošnega končnega avtomata(KA):

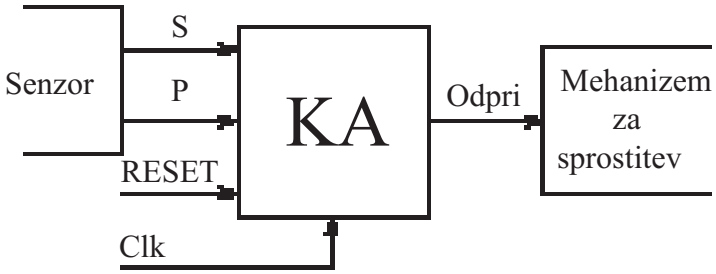
1. Razumevanje problema
2. Abstraktna predstavitev KA
3. Minimizacija stanj KA
4. Kodiranje stanj KA
5. Izbira FF
6. Implementacija KA



Slika 6.4: Primer veljavnosti izhoda.

Primer: Avtomat za žvečilni gumi izstavi žvečilni gumi, ko prejme 150 SIT v bankovcih (S=100 SIT, P=50 SIT). Senzor ločuje med S in P in ne vrača denarja.

1. Razumevanje problema: Shemo vidimo na sliki 6.5

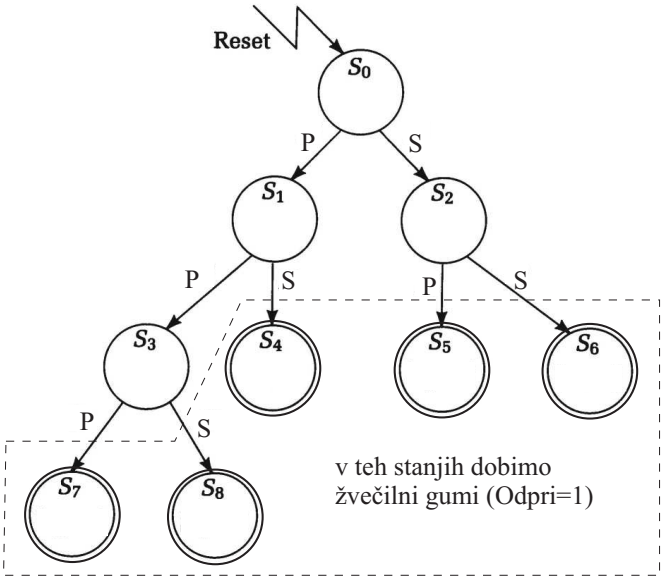


Slika 6.5: Razumevanje problema.

2. Abstraktna predstavitev KA: Pravilne sekvence so:
   P,P,P
   P,P,S
   P,S

S,P
S,S

Od tod sledi osnovni diagram prehajanja stanj, ki ga vidimo na sliki 6.6



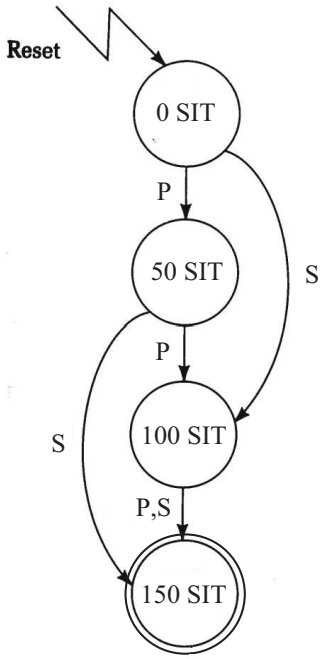
Slika 6.6: Osnovni diagram prehajanja stanj.

3. Minimizacija stanj KA: Če upoštevamo kot stanje sprejet denar in ne tudi način, kako smo ga dobili, lahko poenostavimo diagram. Poenostavljen diagram vidimo na sliki 6.7.

4. Kodiranje stanj KA: Zaenkrat izberimo le 2 bita za 4 stanja in kodirajmo po vrsti:
   0 SIT $\equiv$ 00
   50 SIT $\equiv$ 01
   100 SIT $\equiv$ 10
   150 SIT $\equiv$ 11

   O vplivu kodiranja na obseg logike bomo govorili kasneje (pri optimizaciji KA)

5. Izbira FF: Poljubno. Mi izberemo D FF.



Slika 6.7: Poenostavljen diagram prehajanja stanj

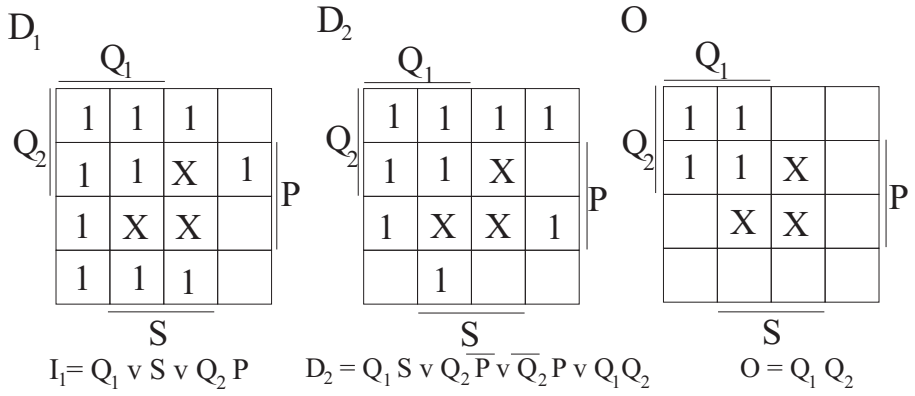
6. Simbolična TPS:

PS	S	P	NS	O
0 SIT	0	0	0 SIT	0
	0	1	50 SIT	0
	1	0	100 SIT	0
	1	1	X	X
50 SIT	0	0	50 SIT	0
	0	1	100 SIT	0
	1	0	150 SIT	0
	1	1	X	X
100 SIT	0	0	100 SIT	0
	0	1	150 SIT	0
	1	0	150 SIT	0
	1	1	X	X
150 SIT	X	X	150 SIT	1

Kodirana TPS:

Q_1(t)	Q_2(t)	S	P	Q_1(t+1)	Q_2(t+1)	O	D_1	D_2
0	0	0	0	0	0	0	0	0
		0	1	0	1	0	0	1
		1	0	1	0	0	1	0
		1	1	X	X	X	X	X
0	1	0	0	0	1	0	0	1
		0	1	1	0	0	1	0
		1	0	1	1	0	1	1
		1	1	X	X	X	X	X
1	0	0	0	1	0	0	1	0
		0	1	1	1	0	1	1
		1	0	1	1	0	1	1
		1	1	X	X	X	X	X
1	1	X	X	1	1	1	1	1

7. Sedaj iz zgornje tabele narišemo Veitcheve diagrame in izpišemo funkcijo. Glej sliko 6.8.



Slika 6.8: Končna rešitev.

Od tod sledi logična shema, ki jo opuščamo.

6.2 Alternativne predstavitve končnih avtomatov

• Podali bomo dve alternativni predstavitvi, ki imata podobnost s programskimi jeziki.

6.2.1 Postopkovni avtomat (ASM)

• Postopkovni avtomat (Algorithmic State Machine - ASM) spominja na diagrame poteka ("flow-chart").

• Slaba stran diagrama prehajanja stanj (DPS) oziroma grafa je, da slabo opisuje strukturo algoritma.

• ASM sestoji iz treh elementov:

1. Stanjski predal ("State Box")

2. Odločitveni predal ("Decision Box")

3. Izhodni predal ("Output Box")

Vsak ASM blok (predal) sestoji iz stanjskega predala in poljubno iz mreže pogojnih in izhodnih predalov. KA se v stabilnem času znotraj "state-time" nahaja natanko v enem stanju oziroma ASM bloku.

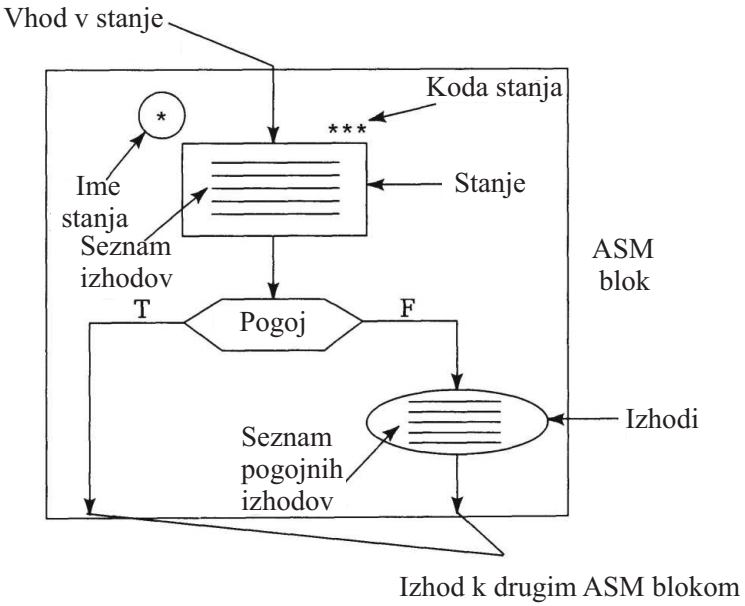
• Primer ASM bloka vidimo na sliki 6.9.

Obstajata 2 izhodna seznama, eden vezan za stanje (Moore-ovi izhodi), eden pa vezan na stanje + pogoj (Mealy - jevi izhodi). Izhodne spremenljivke imajo prefix H oziroma L glede na logiko (active H oziroma active L). Prefix I pomeni, da je izhod aktiviran "Immediately", brez prefixa pomeni, da je zakasnjen do naslednjega urinega dogodka.

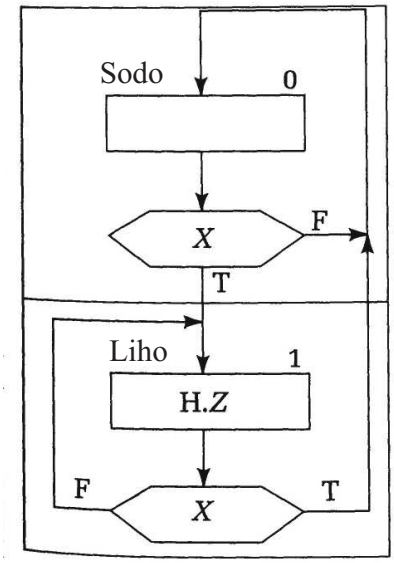
• V primeru več pogojnih predalov vrstni red ni pomemben (komutativnost).

Primer: ASM za kontrolo prioritete (lihe). Glej simbolično tabelo prehajanja stanj v poglavju 5.1 in sliko 6.10.

Primer: Končni avtomat za žvečilni gumi vidimo na sliki 6.11



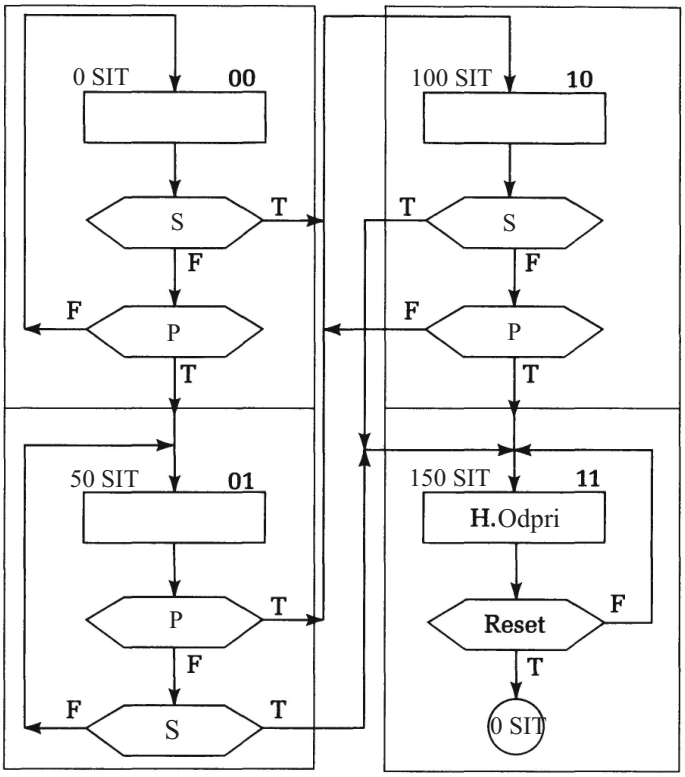
Slika 6.9: Primer ASM bloka.



Slika 6.10: ASM za kontrolo lihe paritete.

6.2.2 Opisni jezik VHDL

VHDL (Very large scale integrated circuits Hardware Description Language) in Verilog sta standardna jezika za opis digitalnih vezij. Takim jezikom rečemo



Slika 6.11: ASM za žvečilni gumi.

tudi jeziki HDL (Hardware Description Languages). VHDL je industrijski standard in ima vse lastnosti visokih programskih jezikov. Detajlni opis jezika presega okvirje tega predmeta. Podali bomo le primer opisa kontrole paritete z VHDL. Tudi VHDL omogoča opis delovanja KA, ki ga lahko po opisu tudi simuliramo.

Vsak VHDL opis ima 2 komponenti:

1. entiteto (vmesnik oz. 'interface' - "entity")
2. arhitekturo (opis delovanja - "architecture")

Prva definira vhodno/izhodne povezave, druga pa opisuje obnašanje KA.

Eden osnovnih tipov jezika VHDL je `bit`, ki ima logični vrednosti '0' in

'1'. Namesto `bita` pa se običajno uporablja tip `std_logic`, ki predstavlja razširjavo; vključuje namreč še nekaj drugih možnih vrednosti, kot so neznana vrednost ('X'), stanje visoke impedance ('Z'), redundanca ('-') ter še nekaj drugih.

Prvi dve vrstici praktično vsakega VHDL programa sta

```
library ieee;
use ieee.std_logic_1164.all;
```

Gre za navedbo standardne knjižnice.

KA običajno opišemo z 2 t.i. procesoma. Proces je konstrukt, ki se evaluira vsakič, ko pride do spremembe kateregakoli izmed signalov, na katere je dani proces občutljiv (ang. 'sensitivity list').

- Prvi proces je t.i. sekvenčni (oznaka npr. `seq`). Le-ta določa začetno stanje KA (ob brisanju) in vpis novega stanja v pomnilne celice stanjskega registra ob fronti ure (`rising_edge(clk)` .. prednja fronta).
- Drugi proces pa je t.i. kombinacijski (oznaka npr. `com`). Ta proces pa opisuje tabelo prehajanja stanj in izhode avtomata.

```
library ieee;
use ieee.std_logic_1164.all;

entity avtomat_pariteta is
port( reset, clk, x: in std_logic;
        y: out std_logic);
end;

architecture arc of avtomat_pariteta is
   type tip_stanje is (S0, S1);
   signal st, nst : tip_stanje; -- stanje, naslednje stanje
begin
seq: process (clk, reset)
    begin
     if reset = '1' then
```

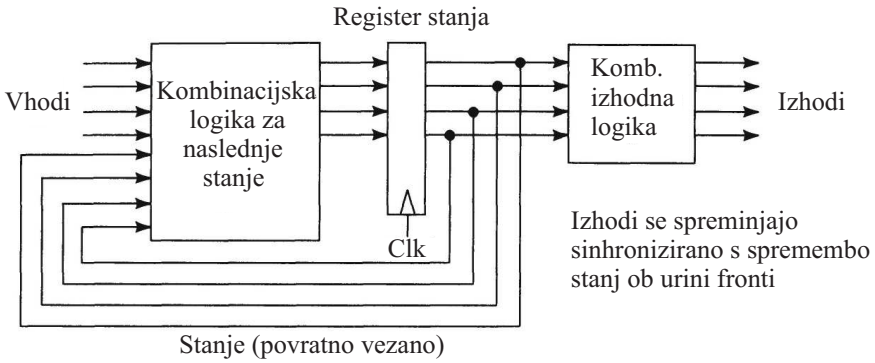
```
      st <= S0;
    elsif rising_edge(clk) then
      st <= nst;
    end if;
  end process seq;

com: process (st, x)
   begin
       case st is
           when S0 => y <= '0';
               if x = '1' then
                   nst <= S1;
               else
                   nst <= S0;
               end if;
           when S1 => y <= '0';
               if x = '1' then
                   nst <= S0;
               else
                   nst <= S1;
               end if;
       end case;
   end process com;
end arc;
```

6.3 Sekvenčna stroja MEALY / MOORE

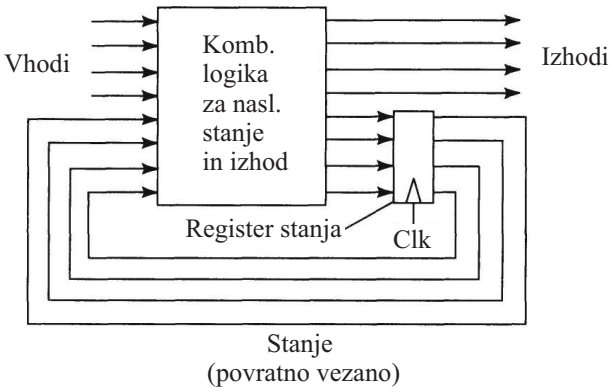
• Izhod pri MOORE-ovem stroju zavisi le od stanja (slika 6.12), pri MEALY-jevem stroju pa od stanja in vhodov (slika 6.13).

• Moorovi izhodi so sinhronizirani z uro, Mealyjevi izhodi pa so asinhronski. To daje prednost Moorovim KA, čeprav obstaja tudi sinhronizirana Mealy-jeva izvedba KA.

• Mealy-jev KA v splošnem rabi manj stanj kot Moorov KA.
Primer: KA, ki daje na izhodu 1, ko zazna na vhodu najmanj dve zaporedni enici (glej sliko 6.14).



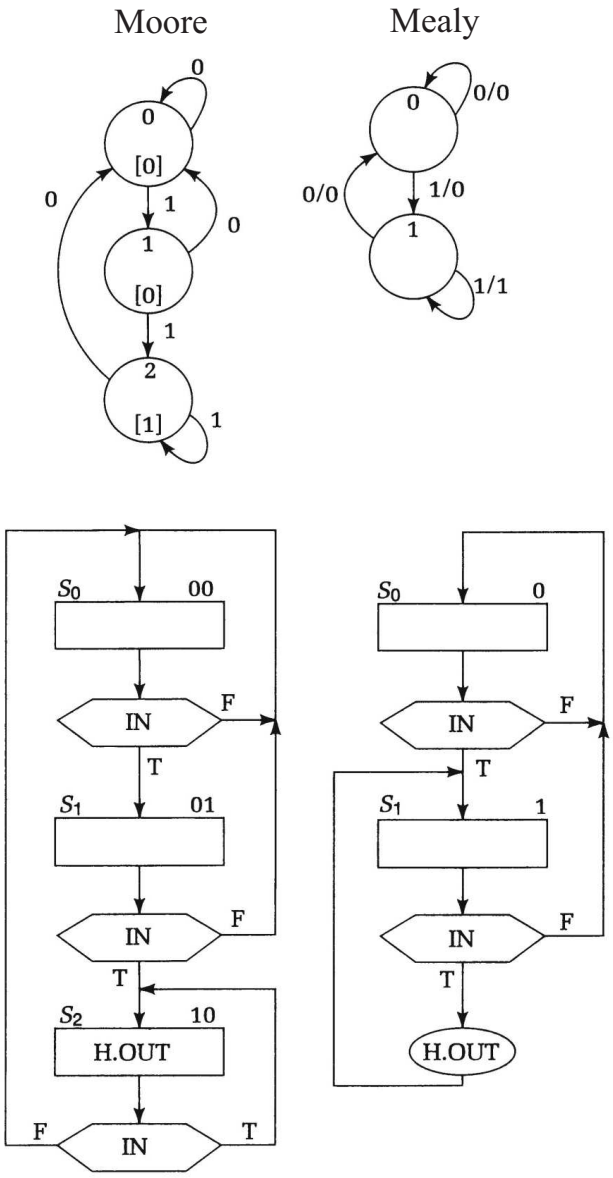
Slika 6.12: Moore.



Slika 6.13: Mealy.

• Sinhronski Mealy-jev stroj: Dosežemo ga z uvedbo dodatnega registra na izhodni poti, tako kot kaže slika 6.15.

• Z dodatnim registrom smo dosegli, da na izhode vplivajo spremembe dodatnega registra (izhodi) ob uri.

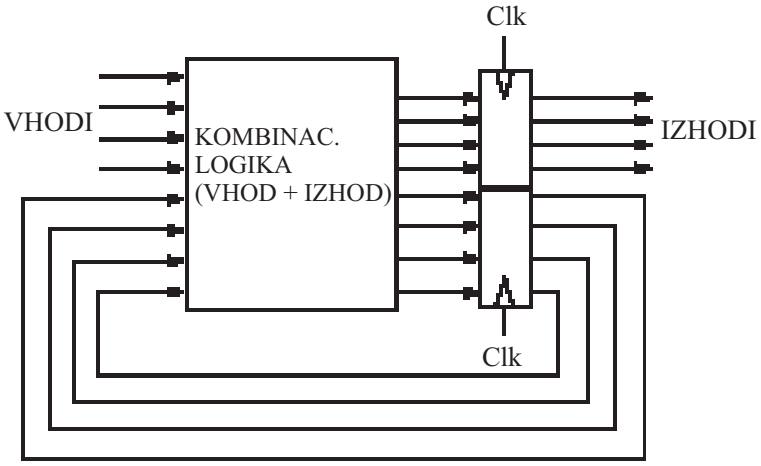


Slika 6.14:

6.4 Optimizacija končnih avtomatov

6.4.1 Minimizacija/redukcija stanj

• Implikacijski (trikotni) diagram vidimo na sliki 6.16: Opisuje vse možne pare stanj, eno stanje podaja abscisa, drugo pa ordinata.



Slika 6.15: Sinhronski Mealy-jev stroj

• Na mestu preseka vpišemo v vsaki vrstici (ki ustreza vhodni kombinaciji) stanje v času (t+1). Če opazujemo par stanj S_i,S_j, ki imata različne izhode, vpišemo X. Vpisana stanja v (t+1) so implicirana stanja (pari).

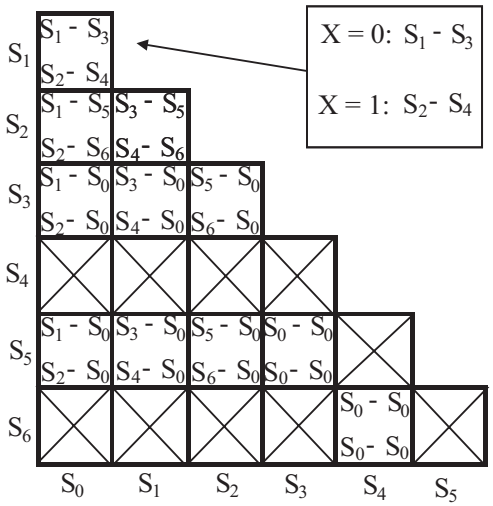
• Sledi eliminacija parov, ki niso ekvivalentni - ali imajo X ali pa pare, ki imajo X. Eliminirane pare prekrižamo (z **X**).

• Ob koncu identična stanja vodijo k istemu stanju in imajo iste izhode.

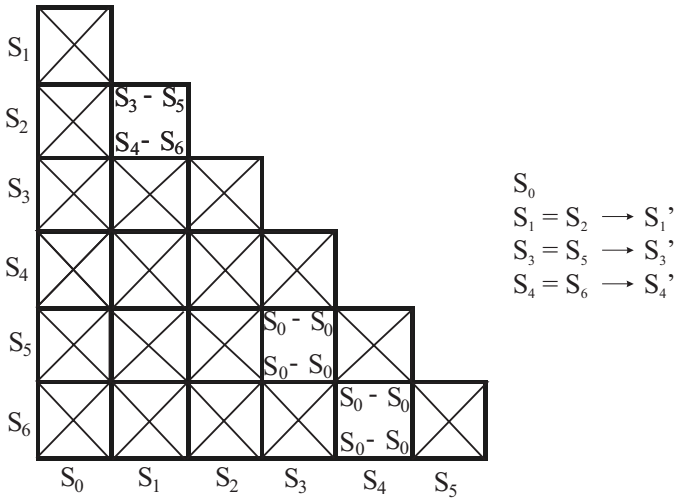
PRIMER: Končni avtomat, ki daje na izhodu 1, ko zazna na vhodu sekvenco 010 ali 110. Kanonska tabela avtomata je naslednja:

		stanje(t+1)/izhod	
vhodna sekvenca	stanje(t)	x=0	x=1
RESET	S_0	$S_1/0$	$S_2/0$
0	S_1	$S_3/0$	$S_4/0$
1	S_2	$S_5/0$	$S_6/0$
00	S_3	$S_0/0$	$S_0/0$
01	S_4	$S_0/1$	$S_0/0$
10	S_5	$S_0/0$	$S_0/0$
11	S_6	$S_0/1$	$S_0/0$

Implikacijski diagram je na sliki 6.16. Po redukciji ekvivalentnih parov dobimo



Slika 6.16: Implikacijski diagram.



Slika 6.17: Minimiziran implikacijski diagram.

diagram, ki ga prikazuje slika 6.17. Velja $S_1 = S_2$ (novo stanje S_1'), $S_3 = S_5$ (novo stanje S_3') in $S_4 = S_6$ (novo stanje S_4'), S_0 pa ni združljivo z nobenim drugim stanjem. S tem se postopek zaključi. Rezultat:

		stanje(t+1)/izhod	
vhodna sekvenca	stanje(t)	x=0	x=1
RESET	S_0	S_1'/0	S_1'/0
$0 \vee 1$	S_1'	S_3'/0	S_4'/0
$00 \vee 10$	S_3'	S_0/0	S_0/0
$01 \vee 11$	S_4'	S_0/1	S_0/0

Za končni rezultat stanja še preimenujemo:

	stanje(t+1)/izhod	
stanje(t)	x=0	x=1
S_0	S_1/0	S_1/0
S_1	S_2/0	S_3/0
S_2	S_0/0	S_0/0
S_3'	S_0/1	S_0/0

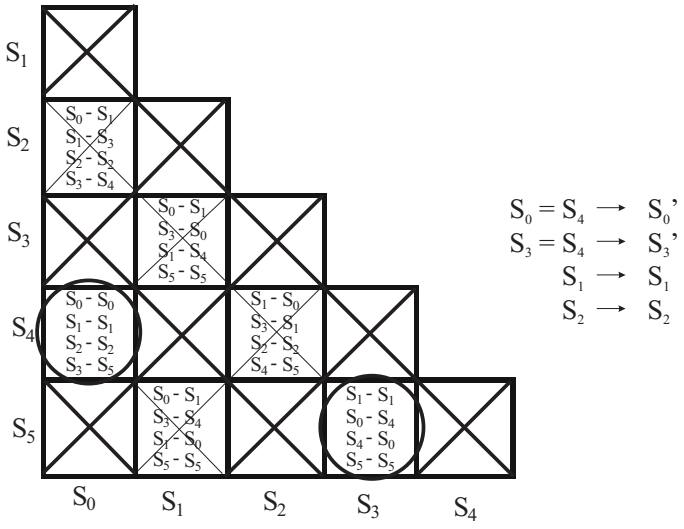
PRIMER: Reducirajte naslednjo kanonsko tabelo:

	00	01	10	11	Izhod
S_0	S_0	S_1	S_2	S_3	1
S_1	S_0	S_3	S_1	S_5	0
S_2	S_1	S_3	S_2	S_4	1
S_3	S_1	S_0	S_4	S_5	0
S_4	S_0	S_1	S_2	S_5	1
S_5	S_1	S_4	S_0	S_5	0

Potek vidimo na sliki 6.18.

Rezultat pa izgleda takole:

	00	01	10	11	izhod
S_0'	S_0'	S_1	S_2	S_3'	1
S_1	S_0'	S_3'	S_1	S_3'	0
S_2	S_1	S_3'	S_2	S_0'	1
S_3'	S_1	S_0'	S_0'	S_3'	0

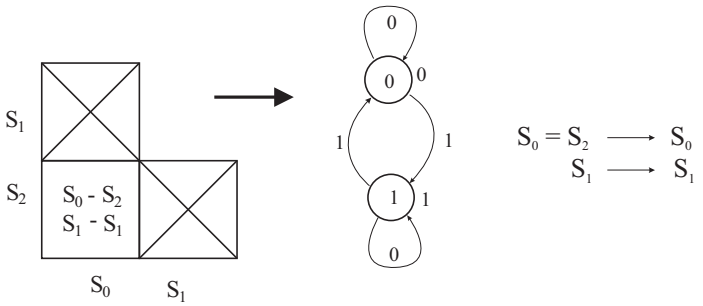


Slika 6.18:

Na koncu lahko stanja še preimenujemo:

	00	01	10	11	izhod
S_0	S_0	S_1	S_2	S_3	1
S_1	S_0	S_3	S_1	S_3	0
S_2	S_1	S_3	S_2	S_0	1
S_3	S_1	S_0	S_0	S_3	0

PRIMER: Kontrola paritete (glej sliko 6.19).



Slika 6.19: Kontrola paritete

	x=0	x=1	Izhod
S_0	0	1	0
S_1	1	2	1
S_2	2	1	0

6.4.2 Kodiranje stanj / izbira FF

• Dobro kodiranje zahteva uporabo obsežnih računalniških orodij, ki zagotavljajo ustrezno redukcijo krmilne logike.

• Ročne metode pa uporabljajo "hevristiko", s katero želimo reducirati Hammingovo razdaljo med sosednjimi stanji.

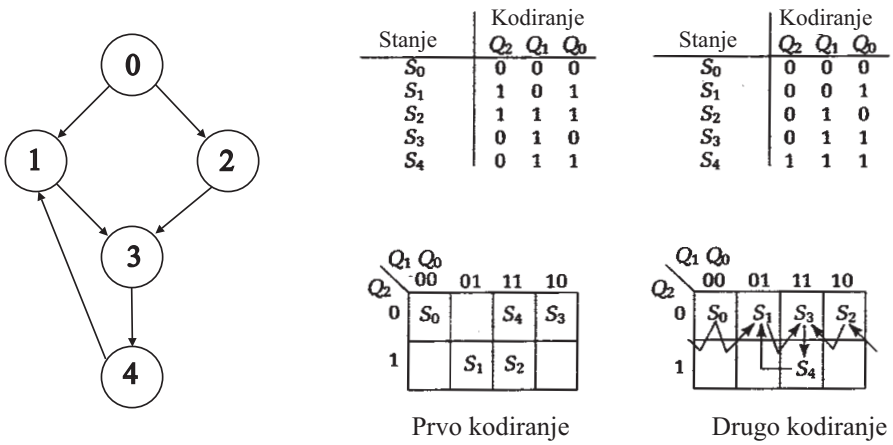
• Ogledali si bomo 2 hevristiki in osnovna vodila pri kodiranju stanj.

A. Hevristika "Preslikava stanj" ("State map"): Opazujemo prehode med stanji in sosednja stanja poizkušamo kodirati čim bliže skupaj (v Hammingovem smislu):

	1 koda			2 koda		
	Q_2	Q_1	Q_0	Q_2	Q_1	Q_0
S_0	0	0	0	0	0	0
S_1	1	0	1	0	0	1
S_2	1	1	1	0	1	0
S_3	0	1	0	0	1	1
S_4	0	1	1	1	1	1

DPS in izpis tabele v Veitchev diagram prikazuje slika 6.20.

B. Hevristika "Najmanjša sprememba bitov": Tukaj opazujemo število bitnih sprememb pri vseh prehodih v KA in izberemo kodo z najmanjšim številom sprememb. NPR.:



Slika 6.20: Hevristika "preslikava stanj".

	1 koda	2 koda
Prehod	število sprememb	število sprememb
$S_0 \rightarrow S_1$	2	1
$S_0 \rightarrow S_2$	3	1
$S_1 \rightarrow S_3$	3	1
$S_2 \rightarrow S_3$	2	1
$S_3 \rightarrow S_4$	1	1
$S_4 \rightarrow S_1$	2	2
$\sum$	13	7

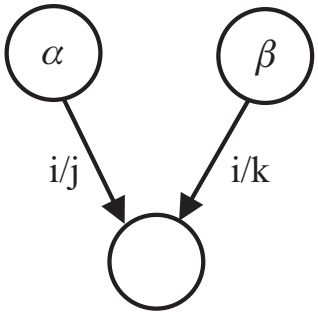
C. Napotki za kodiranje glede na prehajanje stanj oziroma I/O preslikave:

1. Najvišja prioriteta: Če stanji α in β prehajata ob isti črki i v isto stanje, ju poizkušamo kodirati čim bolj skupaj (kot sosednji stanji). Primer je na sliki 6.21.

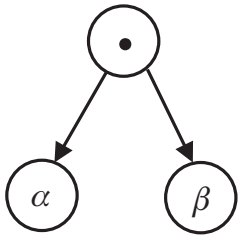
2. Srednja prioriteta: Naslednji stanji α, β istega stanja $(\cdot)$ poizkušamo kodirati kot sosednji stanji. Primer je na sliki 6.22

3. Nizka prioriteta: Stanji α in β z istim izhodom (j) pri istem vhodu (i) poizkušamo kodirati kot sosednji stanji. Primer je na sliki 6.23

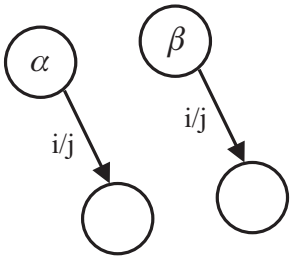
Primer: Detektor sekvence dolžine 3 (glej poglavje 5.4.1) in sliko 6.24:



Slika 6.21: Najvišja prioriteta



Slika 6.22: Srednja prioriteta

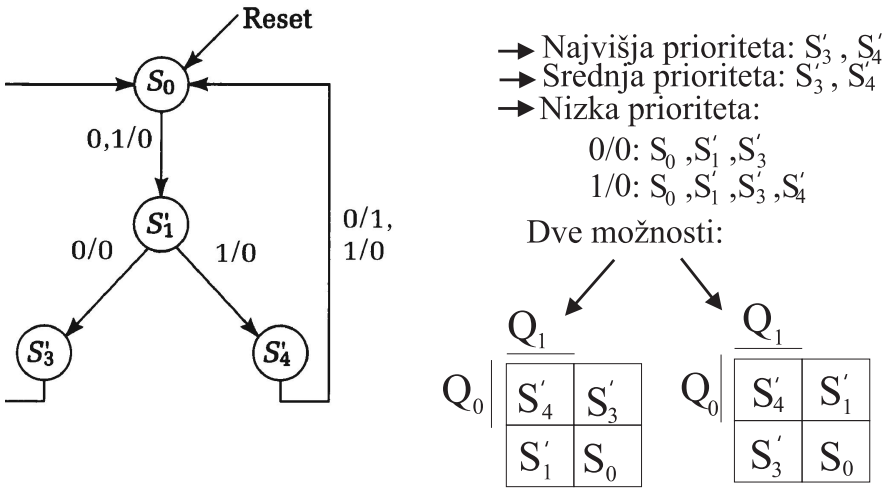


Slika 6.23: Nizka prioriteta

• Običajno začnemo z $S_0 = 0 \ldots 0$, nato pa sledimo prioritetam.

• Redukcijo prehodov omogoča tudi takoimenovano "one hot" kodiranje, kjer ima vsako stanje svoj FF, ki je v lastnem stanju aktiven, sicer pa pasiven.

• Pri izbiri FF se običajno odločamo med JK in D FF. Izbira JK teži k redukciji števila vrat na račun povečanega števila povezav. D FF pa poenostavi proces implementacije in je primeren za VLSI izvedbo, kjer so povezave pomembnejše od vrat.



Slika 6.24: Detektor sekvence dolžine 3.

6.4.3 Delitev končnih avtomatov

• Delitev končnih avtomatov na manjše končne avtomate je smiselna, če je krmilna logika preveč kompleksna. Ideja pri delitvi je, da se vpeljejo "prosta" stanja, s čimer je mogoče reducirati število konjunktivnih izrazov v funkcijah prehajanja stanj, kar je problem pri PAL vezjih (fiksno število konjunkcij).

PRIMER: DPS za KA, ki ga želimo deliti na 2 dela, kot kaže črta narisana preko avtomata na sliki 6.25.

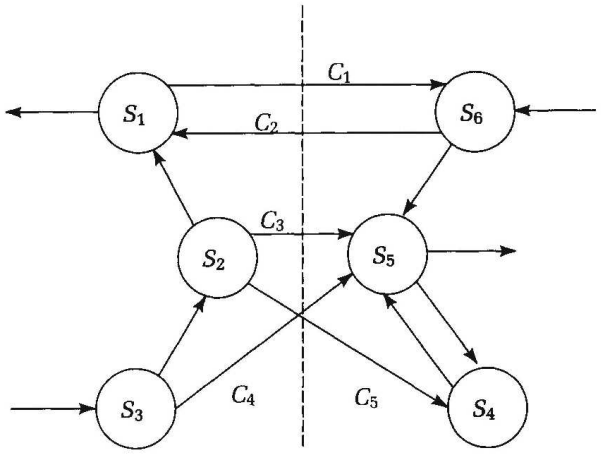
Na vsaki strani dodamo stanje, s pomočjo katerega sinhroniziramo aktivnost obeh novih KA. Ko je avtomat v "prostem" stanju, pomeni, da je aktiven drug KA. Dobljena KA_1 in KA_2 imata $DPS_{1,2}$, kot kaže slika 6.26.

• Pravila za delitev:

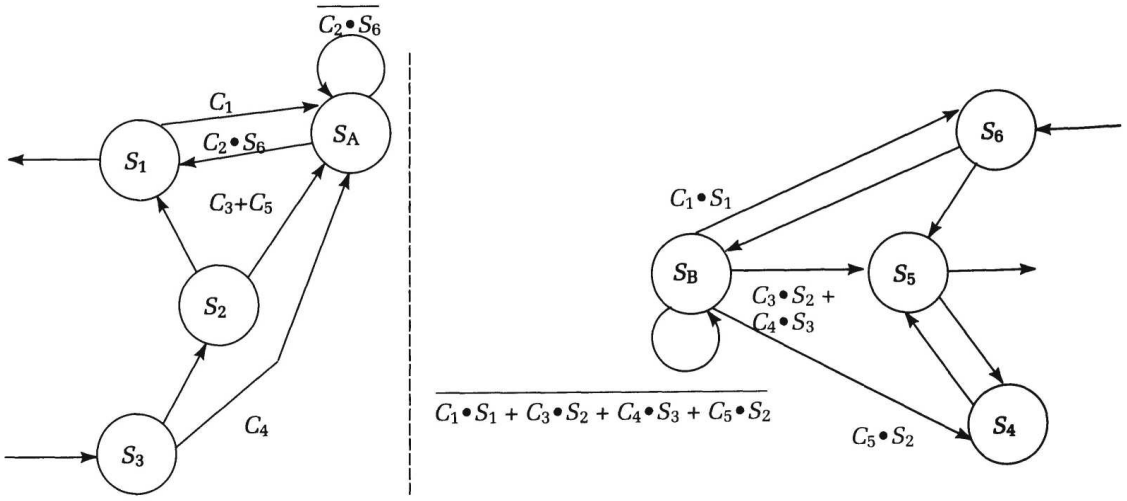
1. Transformacija izvornega stanja (ki prečka mejo dveh avtomatov) je prikazana na sliki 6.27.

2. Transformacija ponornega stanja (Pri prečkanju meje) je prikazana na sliki 6.28.

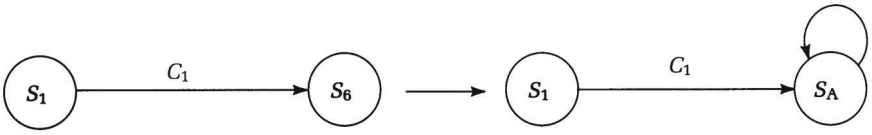
3. Več prehodov meje si deli isti izvor ali ponor, kar vidimo na sliki 6.29.



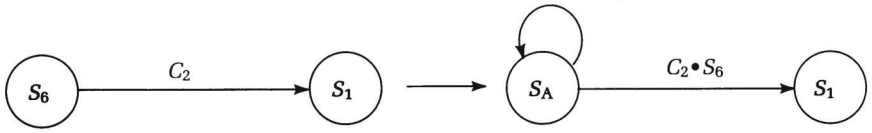
Slika 6.25: Delitev KA. C ... pogoji za prehod med dvema KA.



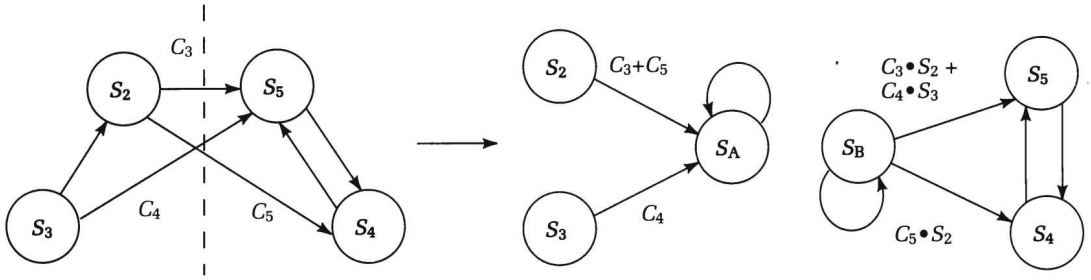
Slika 6.26: DPS za KA_1 in KA_2.



Slika 6.27: Transformacija izvornega stanja.

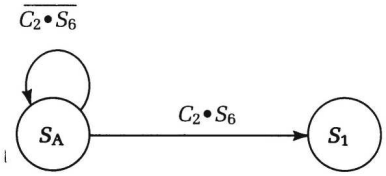


Slika 6.28: Transformacija ponornega stanja.



Slika 6.29: Delitev istega ponora ali izvora.

4. Pogoj ohranjanja prostega stanja je prikazan na sliki 6.30.

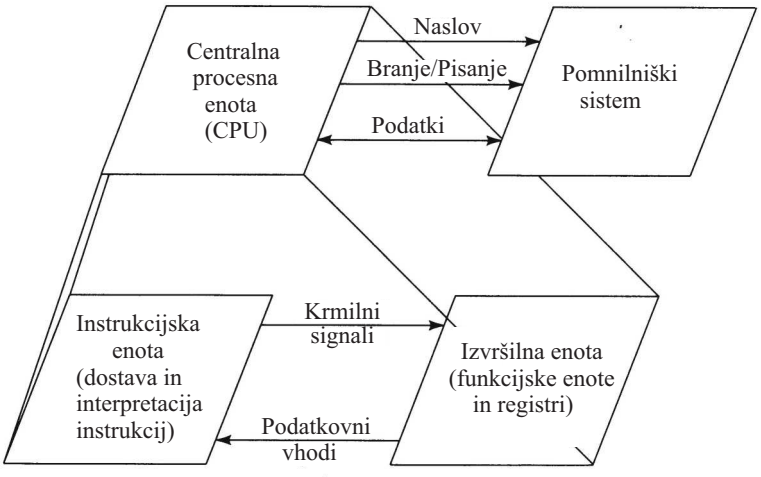


Slika 6.30: Ohranjanje prostega stanja.

Poglavje 7

ORGANIZACIJA RAČUNALNIKA

Računalnik z vgrajenim programom sestoji iz procesne enote in pomnilniškega sistema. Ukazi ali inštrukcije, s katerimi povemo procesorju, kaj želimo od njega, so spravljeni v obliki programa v pomnilniku, skupaj s podatki. Procesna enota je sestavljena iz izvršilne enote (angl. datapath) in krmilne enote (angl. control unit). Slika 7.1 podaja osnovno strukturo računalnika.



Slika 7.1: Osnovna organizacijska struktura računalnika.

Izvršilna enota vsebuje pomembne registre, ki hranijo podatke med izvajanjem inštrukcij in funkcijske enote, n.pr. aritmetično-logično enoto (ALU), pomikalne registre, itd. Krmilna enote je razširjen končni avtomat, ki s prehodi preko svojih stanj 1. dostavlja naslednjo inštrukcijo iz pomnilnika, 2. dekodira ozr. interpretira inštrukcijo in 3. izvede inštrukcijo z ustrezno operacijo nad specificiranimi podatki ter shrani rezultat.

Ključno v procesu snovanja izvršilne enote je, kako povezati različne komponente med seboj, da pri tem zmanjšamo na minimum kompleksnost realizacije in število krmilnih stanj za izvedbo inštrukcij.

V tem poglavju bomo spoznali strukturo krmilnega avtomata in različne načine povezovanja aparaturnih komponent med seboj. V nadaljevanju pa si bomo ogledali različne načine realizacije krmilne enote računalnika.

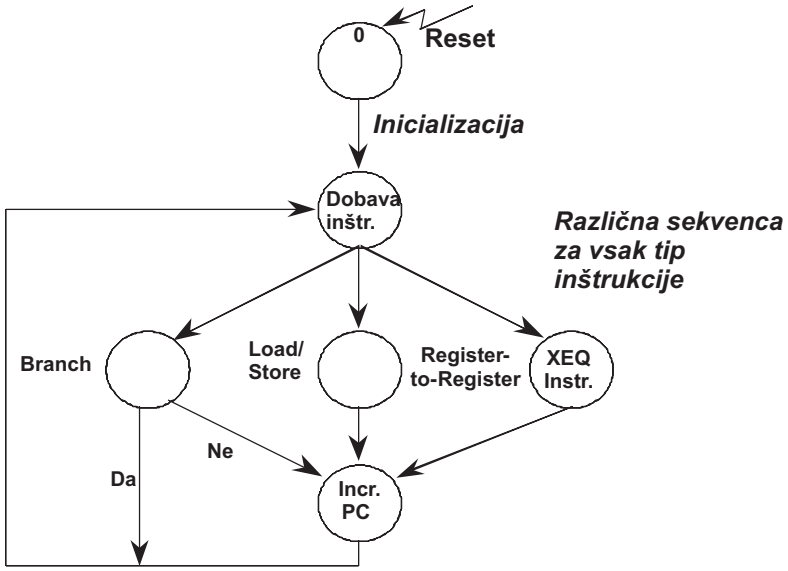
7.1 Struktura krmilnega avtomata

Procesna krmilna enota je precej bolj kompleksna kot običajni končni avtomati, ki smo jih spoznali do sedaj. Krmilna enota potrebuje dostop do vseh komponent izvršilne enote, do vseh njenih registrov, po drugi strni pa mora nadzorovati izvajanje vseh inštrukcij, ki na različne načine uporabljajo omenjene resurse.

Krmilna enota je lahko v eni od štirih osnovnih faz: Resetiranje (angl. Reset), Dostava naslednje inštrukcije (angl- Fetch the Next Instruction), Dekodiranje inštrukcije (angl. Decode the Instruction) in Izvršitev inštrukcije (angl. Execute the Instruction). Visokonivojski diagram poteka za krmilno enoto je predstavljen na Sliki 7.2.

Na zgornji sliki so podane tri paralelne veje, ki ustrezajo trem različnim tipom inštrukcij. Enega predstavljajo skočne inštrukcije (angl. branch), drugega inštrukcije naloži/shrani (angl. Load/Store), tretjega pa inštrukcije register-register. Seveda ima vsaka inštrukcija v svoji veji večje ali manjše število stanj krmilnega avtomata, ki določajo preko izhodnih krmilnih spremenljivk, kaj se mora v danem trenutku izvajati v zvezi s komponentami izvršilne enote.

Zunanji signal 'reset' postavi končni avtomat (KA) krmilne enote v začetno stanje (angl. Reset state), ki pomeni inicializacijo procesorja. To pomeni



Slika 7.2: Visokonivojski diagram poteka krmilnega avtomata.

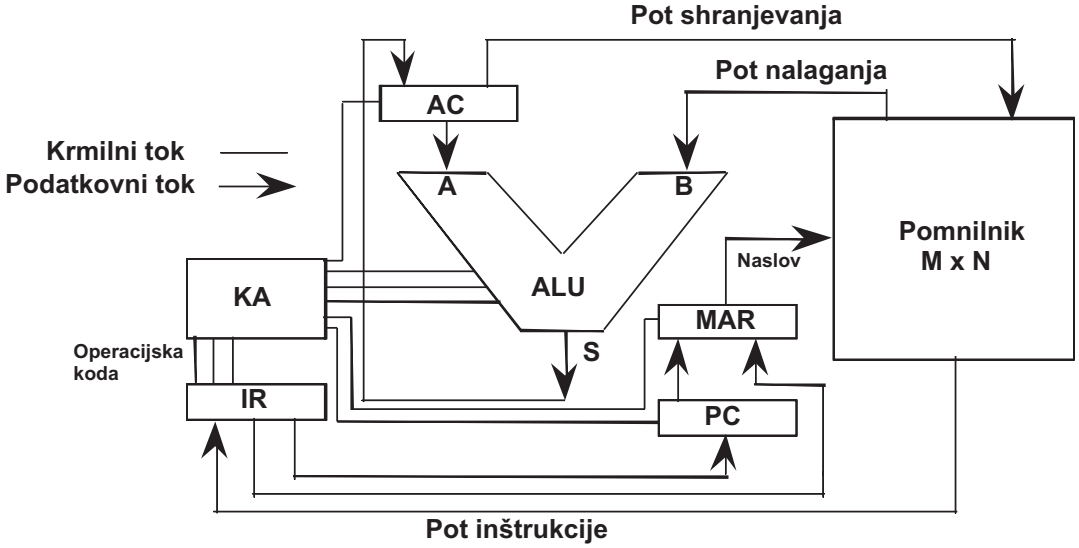
inicializacijo še drugih registrov, n.pr. registra PC (programski števec), ki hrani naslov naslednje inštrukcije, registra AC (akumulator, ki hrani običajno operand in rezultat), itd. Čeprav je na sliki inicializacija podana z enim samim stanjem, pa je ta proces lahko izveden preko sekvence stanj KA.

Naslednja faza je dostava nove inštrukcije. Vsebina PC naslovi pomnilnik in njegova vsebina se prenese v inštrukcijski register IR. Tudi ta faza je podana na Sliki 7.2 z enim stanjem, čeprav njena dejanska implementacija zahteva sekvenco preko sekvence stanj KA.

Sledi dekodiranje vsebine IR registra, ki hrani novo inštrukcijo. To pomeni odločitev, po kateri veji bo izvajanje potekalo, v primeru poeostavljene zgornje slike, ali po veji 'Branch', 'Load/Store' ali pa po veji 'Xeq', ki pomeni računski tip inštrukcije. Vsaki inštrukciji zopet ustreza določeno sekvenco stanj KA, s katero nadzira njeno izvrševanje.

Ob koncu je potrebno pripraviti pogoje za naslednjo inštrukcijo. To pomeni, ali povečati PC za eno lokacijo v pomnilniku, kjer se nahaja naslednja inštrukcija, ali pa vpisati v PC nov naslov, ki je posledica inštrukcije, ki testira določeno vrednost v registru (običajno v AC) in ki zahteva nadaljevanje programa glede na rezultat testa.

Na Sliki 7.3 je prikazan blok diagram računalnika, na katerem so podani njegovi osnovni resursi in vodila, ki jih povezujejo. Povezave s puščicami označujejo podatkovne poti, povezave brez njih pa krmilne tokove, preko katerih KA nadzira izvajanje inštrukcij.



Slika 7.3: Blok diagram računalnika.

Na zgornji sliki najdemo registre IR, AC, PC ter naslovni ('address') register pomnilnika MAR, nadalje KA, ALU in pomnilnik M. Jedro izvršilne enote predstavlja ALU enota in akumulator AC. AC je izvor ali ponor večine prenosov podatkov. Tipična inštrukcija je simbolično podana z zapisom:

AC := AC <op> M

ki pomeni naslednje: izvede se operacija <op> nad operandoma, od katerih eden je v AC, drugo v pomnilniku M, rezultat pa se shrani v AC. Pred to inštrukcijo je torej potrebno shraniti en operand v AC, kar napravimo z inštrukcijo tipa 'Load/Store', konkretno z inštrukcijo 'Load', ki pomeni zapis vsebine pomnilnika (naslov je podan v inštrukciji) v akumulatorski register AC. Omenjeni naslov operanda se iz IR registra prenese v MAR, nato pa sledi ukaz za čitanje vsebine iz pomnilnika M. Pri tem se operand običajno izpiše najprej v vmesni register MBR ('Memory Buffer Register'), ki na Sliki 7.3 ni narisan zaradi poenostavitve, od tod pa v AC.

Pri izvajanju inštrukcij programa prihaja do številnih prenosov podatkov med procesorjem in pomnilnikom. Temu včasih pravimo tudi ozko grlo procesiranja (angl. bottleneck), saj je zaradi njega procesiranje izrazito sekvenčno in zato počasnejše, kot bi lahko bilo. Glede na povezavo procesorja s pomnilnikom ločimo dve vrsti arhitektur: Princetonska arhitektura, kjer so podatki in programi v istem pomnilniku in Harvardska arhitektura, kjer so podatki shranjeni v enem pomnilniku, programi pa v drugem. Seveda je Princetonska enostavnejša in zato bolj pogosta, Harvardska pa bolj zmogljiva in zato dražja in redkejša. Pri slednji lahko n.pr. shranjevanje rezultatov in dostava naslednje inštrukcije potekata paralelno, zato je izvajanje programa lahko hitrejše.

Za ilustracijo krmilnih signalov in podatkovnih prenosov, ki so potrebni pri izvajanju inštrukcij, si oglejmo preprosto inštrukcijo ADD, ki prišteje vsebino na specificirani pomnilniški lokaciji k vsebini v AC:

1. Inštrukcija mora biti najprej dostavljena iz pomnilnika. To pomeni prenos vsebine PC registra v MAR register in nato proženje operacije čitanja iz pomnilnika. Inštrukcija se nato iz pomnilnika prenese v IR register.

2. Določeni biti v inštrukciji pomenijo operacijsko kodo inštrukcije (angl. op code), v tem primeru kodo ADD inštrukcije. Ti biti predstavljajo del vhodnih spremenljivk v krmilni avtomat KA. Preostali biti v inštrukciji ADD določajo naslov pomnilnika, kjer se nahaja drugi operand.

3. KA povzroči prenos naslova iz IR v MAR register in sproži operacijo čitanja operanda ter njegov prenos na B vhod ALU enote.

4. Sledi krmiljenje ALU enote z binarno kodo, ki ustreza ADD operaciji, ozr. seštevanju A in B vhodov v ALU enoto, ki daje rezultat na S izhod ALU enote (glej Sliko 7.3).

5. KA nadaljuje s proženjem prenosa rezultata seštevanja od S izhoda ALU enote v AC register.

6. Končno KA povzroči povečanje (angl. increment) PC tako, da kaže na naslednjo inštrukcijo v programu. Nato se KA vrne v začetno stanje ozr. k prvemu koraku zgoraj.

Takšen opis izvajanja inštrukcij pa je nepregleden. Zato se uporablja t.im.

prenosna registrska notacija (angl. Register Transfer Notation), s katero bi omenjeno izvajanje inštrukcije ADD podali takole:

Dostava inštrukcije:
PC → MAR;
Memory Read;
Memory → IR;

Dekodiranje inštrukcije:
IF IR <op code> = ADD_FROM_MEMORY THEN

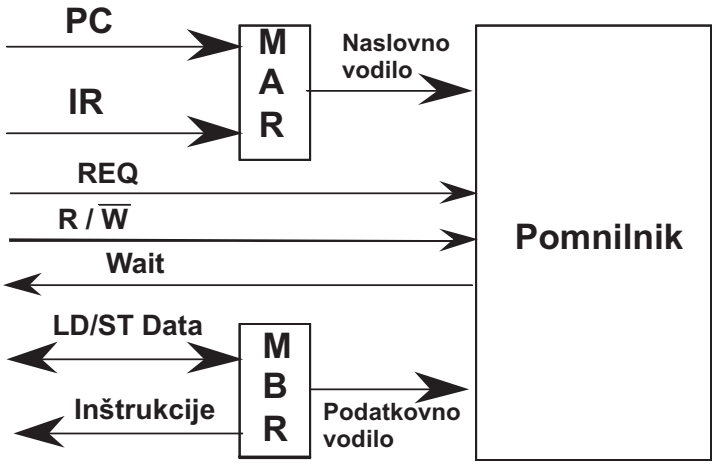
Izvršitev inštrukcije:
IR<address bits> → MAR;
Memory Read;
Memory → ALU B;
AC → ALU A;
ALU ADD;
ALU S → AC;
PC increment;

V zgornjem RTN zapisu so uporabljeni krmilni signali, n.pr. Memory Read, ALU ADD, PC increment, ki pomenijo proženje ustreznih operacij. Oznaka izvor → ponor pomeni prenos podatkov od izvora k ponoru. Bolj podrobno je mogoče opisati delovanje procesorja s pomočjo RTD zapisa (angl. Register Transfer Description), ki ga bomo spoznali v nadaljevanju.

7.2 Pomnilniški vmesnik

Pomembno vlogo pri izvajanju inštrukcij igra pomnilniški vmesnik, saj se preko njega prenašajo inštrukcije in podatki. Njegove bistvene elemente podaja Slika 7.4.

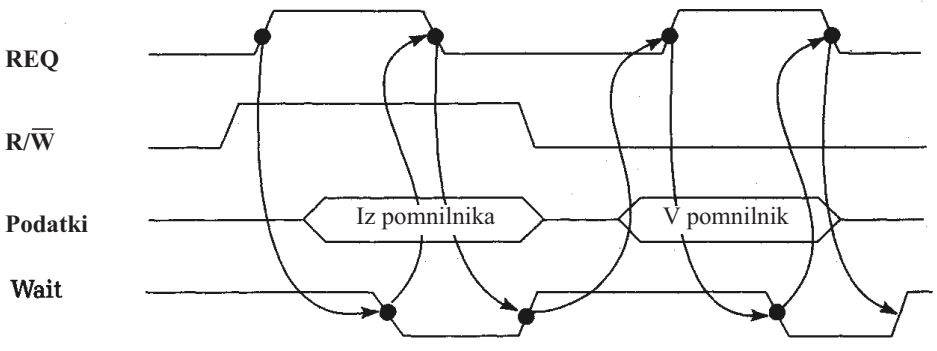
Zgornja slika podaja vmesnik za Princetonsko arhitekturo, kjer sta bistvena elementa MAR in MBR register ter trije krmilni signali: REQ, READ/$\overline{\text{WRITE}}$ in WAIT. Kot je razvidno iz slike, služi MAR register za naslavljanje pomnilnika in je zato prenos preko njega enosmeren. MBR pa je namenjen prenašanju podatkov, zato je prenos preko njega dvosmeren. Poleg adresnih in podatkovnih poti, ki gredo preko omenjenih registrov, najdemo v vmesniku še omenjene



Slika 7.4: Pomnilniški vmesnik.

krmilne signale. Signal REQ obvesti pomnilnik, da želi procesor dostop do njega. Signal READ/$\overline{\text{WRITE}}$ specificira, za kakšen dostop gre, ali za vpisovanje v pomnilnik (operacija 'store' ali 'write') ali za čitanje (operacija 'load' ali 'read'). WAIT signal pošilja pomnilnik procesorju in z njim potrdi sprejem zahteve (angl. acknowledge) ozr. ga obvesti, kdaj so podatki pripravljeni (prečitani ozr. vpisani). Omenimo na tem mestu, da smo v poglavju 4 že spoznali dva protokola (s štirimi in z dvema cikloma), ki na osnovi teh krmilnih signalov omogočata potrebno komunikacijo med procesorjem in pomnilnikom.

KA nadzira tudi protokol za izmenjavo podatkov med procesorjem in pomnilnikom. Slika 7.5 podaja časovni diagram krmilnih signalov.

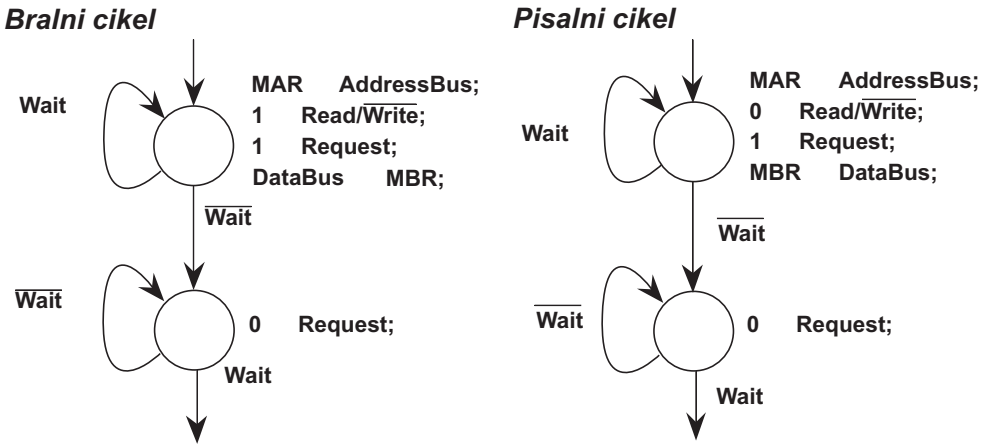


Slika 7.5: Časovni diagram krmilnih signalov pomnilniškega vmesnika.

V časovnem diagramu na zgornji sliki za slučaj štirifaznega protokola pomeni:

1. REQ 0 → 1: Zahteva poslana pomnilniku, podatki na vodilu.
2. WAIT 1 → 0: Zakasnitev odstranjena. Podatki se zapišejo v MBR.
3. REQ 1 → 0: Zahteva odstranjena.
4. WAIT 0 → 1: Zakasnitev postavljena.

Detajl diagrama prehajanja stanj (DPS) krmilnega avtomata KA, ki nadzira čitanje / vpisovanje preko pomnilniškega vmesnika, je podan na Sliki 7.6.



Slika 7.6: Izsek iz DPS za nadzor pomnilniškega vmesnika.

Podani segment DPS predpostavlja Moorovo implementacijo KA. Pri operaciji čitanja ('read') vstopimo v stanje, ki odpre adresno vodilo od MAR k pomnilniku, aktivira READ in REQ signala in zapiše vsebino na podatkovnem vodilu v MBR. Slednji prenos je pravilen le, če ima pomnilnik odstranjeno zakasnitev (WAIT = 0), zato je potrebno vztrajati v tem stanju, dokler omenjeni pogoj ni izpolnjen. Pri izhodu iz stanja, ki ga pogojuje odstranjena zakasnitev, se odstrani zahteva (REQ = 0) ter zapre adresno vodilo. Ob Postavitvi zakasnitve (WAIT = 1) zapustimo še drugo stanje, kar pomeni, da je vmesnik pripravljen na novo zahtevo procesorja po pomnilniku. Operacija zapisovanja ('write') poteka podobno, zato je tudi krmiljenje vmesnika s strani KA podobno.

7.3 Vhodno/Izhodni (V/I) vmesnik

Poleg procesorja in pomnilnika so pogosti prenosi podatkov tudi med procesorjem in vhodno/izhodnimi napravami. Le-te so tipično precej počasnejše od procesorja, zato je pomembno, kako poteka komunikacija z njimi.

V/I naprave so lahko povezane s procesorjem na dva glavna načina: ali preko specialnega V/I vodila ali pa preko skupnega vodila s pomnilnikom. Večina sodobnih računalnikov uporablja drugi način. V tem primeru je prednost v tem, ker ne potrebujemo specialnih inštrukcij za izvedbo V/I operacij. Ta strategija se imenuje pomnilniška preslikava V/I (angl. memory- mapped I/O).

V/I čas dostopa se meri v msek, dostop do pomnilnika pa je manjši od mik/-ro/-se/-kun/-de. Zato ni primerno, da procesor čaka, dokler V/I naprava ne konča svojega dela. Zato je povezovanje procesorja z V/I napravami bolj zapleteno.

Obstajata dva načina, kako bolje izkoristiti procesor v primeru uporabe V/I naprav. Prvi se imenuje skaniranje (angl. polling), drugi pa prekinjanje (angl. interrupt). Pri prvem načinu se stanje V/I naprave naslavlja kot pomnilniški register, ki ga sistemski program periodično opazuje in v primeru zahteve, ustrezno ukrepa. V primeru prekinjanja pa se sproži prekinitev vsakič, ko se V/I operacija konča. To sproži prehod KA v posebno stanje, v katerem se shranijo vsi pomembni registri, procesor nadaljuje delo v ustrezni prekinitveni rutini, ki izvede V/I prenos, ob koncu pa se vrne registrom stare vrednosti in sledi nadaljevanje na mestu, kjer je bilo procesiranje prekinjeno.

Skaniranje se uporablja pri zelo zmogljivih računalnikih, kjer ne sme priti do prekinjanja programov zaradi V/I zahtev, prekinjanje pa pri vseh ostalih računalnikih, n.pr. PC računalnikih in TS sistemih (angl. Time-Sharing). Prekinjanje zahteva manjšo spremembo v osnovnem diagramu poteka, ki ga podaja Slika 7.2. Pred dostavo nove inštrukcije mora procesor preveriti, ali je bil sprejet signal za prekinitev. Če ni bil, sledi normalna dostava, sicer pa sledi posebna sekvenca stanj, ki shrani registre, nato pa sledi normalno izvajanje inštrukcij iz t.im. prekinitvene sekvence (angl. interrupt routine), ki opravi V/I operacijo glede na zahtevo, nato pa ob koncu sledi posebna inštrukcija RTI (angl. ReTurn from Interrupt), s katero se obnovijo stari registri in vrne kontrola na mesto prekinitve.

7.4 Strategije povezav

Obstajajo tri splošne strategije povezovanja resursov med seboj: točkovno povezovanje (angl. point-to-point), enojno deljeno vodilo (angl. single shared bus) in večkratna specializirana vodila (angl. multiple special-purpose buses). Vsaka predstavlja ustrezno vrednotenje med kompleksnostjo realizacije izvršilne in nadzorne enote na eni strani in stopnjo paralelnosti na drugi, ki se kaže v procesorski uspešnosti. Ta je definirana s številom krmilnih stanj (ozr. ciklov ure), ki so potrebni za dostavo in izvršitev tipične inštrukcije. Kadar izvršilna enota pri izvajanju inštrukcije podpira več prenosov hkrati, zahteva krmilna enota manj stanj (in urinih ciklov) za njen nadzor.

V nadaljevanju bomo spoznali tri metode za organizacijo povezav med resursi. Pri tem bomo uporabili kot primer štiri registre, ki predstavljajo resurse, ki jih želimo povezati med seboj. Za oceno metod bo služila operacija SWAP (R_i, R_j), ki zahteva zamenjavo vsebin dveh registrov:

SWAP (R_i, R_j):

$R_i \rightarrow R_j$

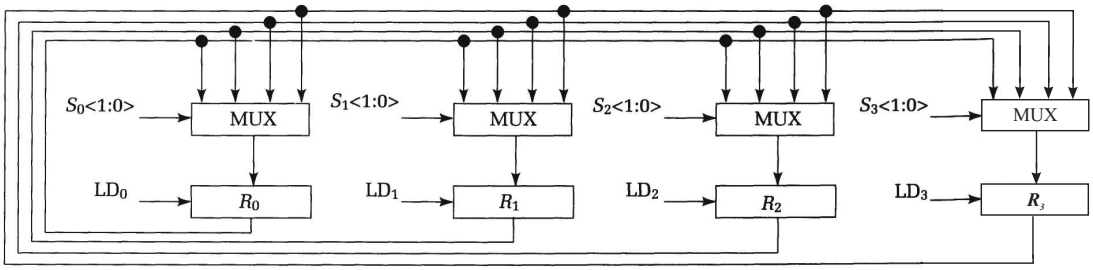
$R_j \rightarrow R_i$

7.4.1 Točkovne povezave (angl. Point-to-Point Connections)

V tem primeru obstaja pot med poljubnim parom resursov. Slika 7.7 podaja točkovno povezavo za primer štirih registrov.

Na zgornji sliki pomeni MX Multipleksor, R_i, $i = 0,...,3$, štiri registre (resursi), S_i<1:0> selektorske vhode (2 bita zaradi štirih registrov), LD_i pa ukaze za vpis (angl. load) v registre. Vsak od štirih registrov lahko hkrati vpiše vsebino iz poljubnega registra. Pri tem predpostavljamo, da se vpis izvede le ob prisotnosti ure. V primeru, da so registri dolžine N bitov, potrebujemo N MX 4/1 za vsak resurs.

Vzemimo primer, ko želimo zamenjati vsebini registrov R_1 in R_2. Krmilni signali so tedaj:



Slika 7.7: Točkovna povezava štirih registrov.

$01 \rightarrow S_2$ <1:0>;

$10 \rightarrow S_1$ <1:0>;

$1 \rightarrow \mathrm{LD}_2$;

$1 \rightarrow \mathrm{LD}_1$;

Vse omenjene krmilne akcije lahko opravimo v istem stanju KA. Zato je takšna rešitev povezav maksimalno hitra, vendar pa zelo draga, saj za n.pr. $N = 32$ to pomeni 160 vrat na register ali 640 vrat za izvedbo točkovne povezave med 4 registri.

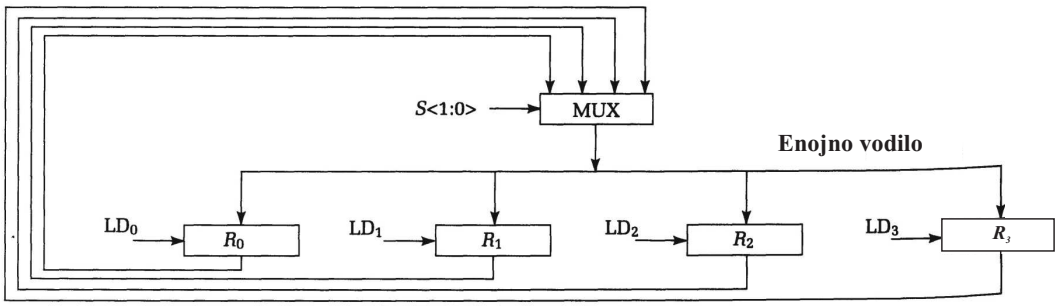
7.4.2 Enojno vodilo (angl. Single Bus)

Pri temu načinu povezav resursov obstaja eno samo vodilo, ki si ga morajo resursi deliti. Sliko povezave za slučaj štirih registrov podaja Slika 7.8.

Oglejmo si dva primera, prvič želimo vpisati v R_0 vsebino iz R_1 in v R_3 vsebino iz R_2, nato pa še operacijo SWAP (R_1, R_2). V prvem primeru potrebujemo dve stanji KA, v drugem pa kar tri:

A. $R_1 \rightarrow R_0, R_2 \rightarrow R_3$:

Stanje X: $(R_1 \rightarrow R_0)$
$01 \rightarrow S$<1:0>;



Slika 7.8: Povezava z enojnim vodilom.

$1 \rightarrow \mathrm{LD}_0$;
Stanje Y: $(R_2 \rightarrow R_3)$
$10 \rightarrow S\text{<}1{:}0\text{>}$;
$1 \rightarrow \mathrm{LD}_3$;

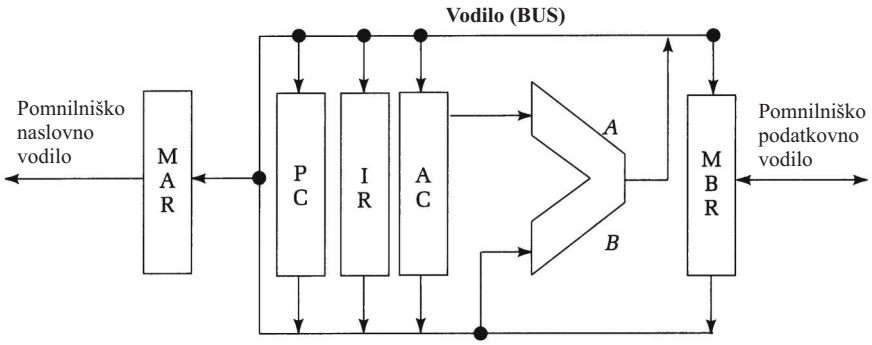
B. SWAP (R_1, R_2):

Stanje X: $(R_1 \rightarrow R_4)$
$001 \rightarrow S\text{<}2{:}0\text{>}$;
$1 \rightarrow \mathrm{LD}_4$;
Stanje Y: $(R_2 \rightarrow R_1)$
$010 \rightarrow S\text{<}2{:}0\text{>}$;
$1 \rightarrow \mathrm{LD}_1$;
Stanje Z: $(R_4 \rightarrow R_2)$;
$100 \rightarrow S\text{<}2{:}0\text{>}$;
$1 \rightarrow \mathrm{LD}_2$;

V slednjem primeru potrebujemo rezervni register, R_4 v zgornjem primeru, ustrezno večji MX in pa seveda tri krmilna stanja. Zato je ta varianta povezav druga skrajnost, saj ni draga, je pa počasna. Kot večkrat obstaja tudi kompromis, ki je tretja varianta povezav, ozr. večvodilna povezava.

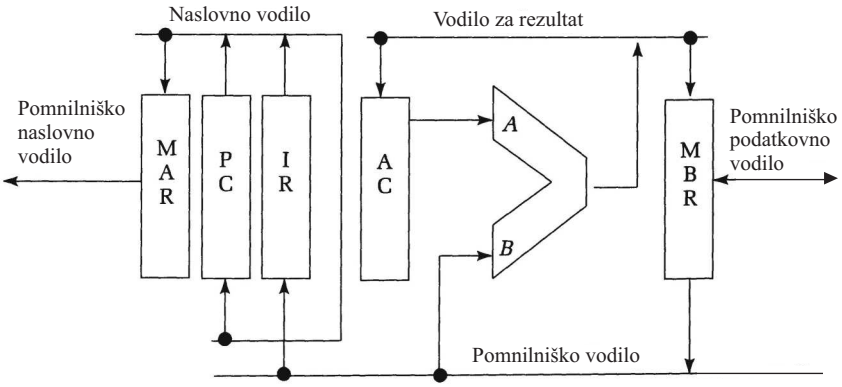
7.4.3 Večvodilna povezava (angl. Multiple Buses)

V tem primeru imamo več kot eno vodilo, vendar manj kot je resursov. Oglejmo si kot izhodišče shemo izvršilne enote procesorja s samo enim vodilom. Prikazuje jo Slika 7.9.



Slika 7.9: Shema izvršilne enote z enim vodilom.

Alternativno shemo ekvivalentne izvršilne enote pa prikazuje Slika 7.10, kjer vidimo tri vodila, adresno, pomnilno (podatkovno) in vodilo za rezultat.



Slika 7.10: Shema izvršilne enote s tremi vodili.

Takšna shema omogoča večjo stopnjo paralelnosti, kar je razvidno iz opisa krmiljenja inštrukcije ADD Mem[X], kar pomeni, da želimo k AC dodati vsebino na lokaciji X v pomnilniku (angl. Memory) in rezultat zapisati v AC.

A. Opis krmiljenja v primeru enega vodila:

Dostava operanda
1. cikel (ura):
   IR<op naslov> → BUS;
   BUS → MAR;
2. cikel:
   Memory Read;
   Databus → MBR;

Izvršitev ADD
3. cikel:
   MBR → BUS;
   BUS → ALU B;
   AC → ALU A;
   ADD;

Zapis rezultata
4. cikel:
   ALU Result → BUS;
   BUS → AC;

B. Pri krmiljenju s tremi vodili potrebujemo le tri cikle, saj je mogoče prekrivanje izvršitve inštrukcije z zapisom rezultata.. V zgornjem primeru je tedaj mogoče izvršitev ADD in zapis rezultata izvesti v enem ciklu, kar je posledica ločenih vodil.

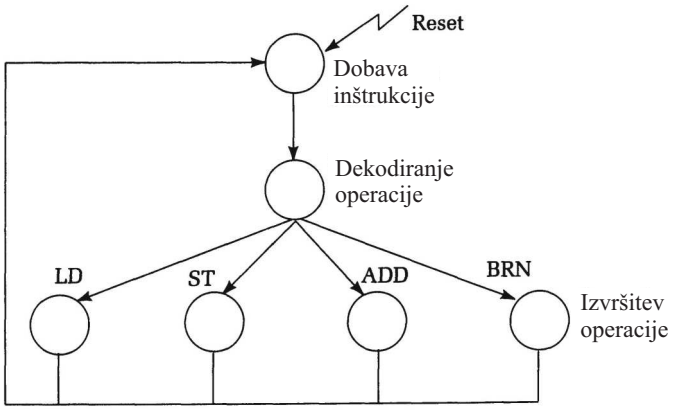
7.5 Krmilni Avtomat za preprosto procesno enoto

V tem poglavju želimo združiti vsebine prejšnjih poglavij tako, da bomo pokazali DPS za KA, ki krmili preprosto izvršilno enoto, ki je sposobna izvajati le štiri operacije:

Load(00): Mem[X] → AC;

Store(01): AC → Mem[X];
ADD(10): AC + Mem[X] → AC;
BRN(11): AC < 0 ⇒ XX → PC;

Predpostavljamo 16-bitno inštrukcijo, kjer sta 2 bita za op kodo, ostali pa za naslov ali adreso. Zgoraj so v oklepajih za imeni inštrukcij podane op kode, nato pa v RTN notaciji opisane njihove naloge. Slika 7.11 podaja visokonivojski DPS za KA tako definiranega procesorja.



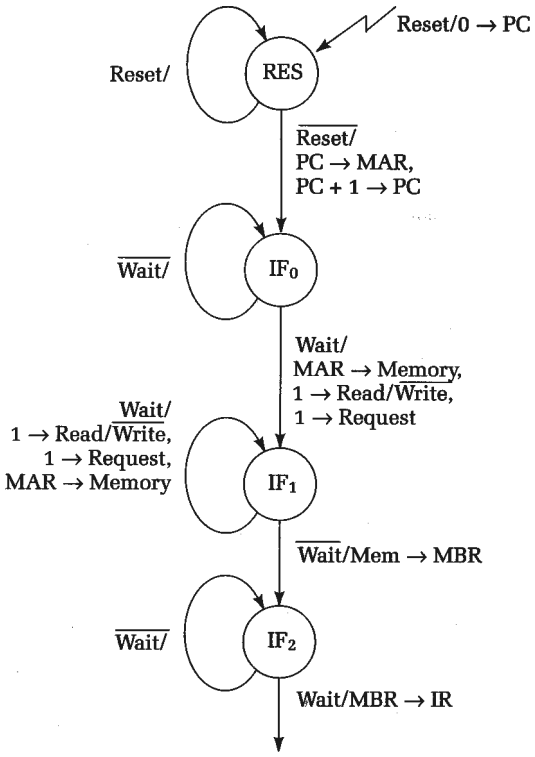
Slika 7.11: Visokonivojski DPS za KA preprostega procesorja.

V nadaljevanju želimo podrobneje definirati DPS za KA podanega procesorja. Ker smo si vse detajle že ločeno ogledali, jih bomo sedaj le še združili v celoto. Odločimo se za Mealyjevo izvedbo KA.

Dostavno fazo skupaj z resetiranjem procesorja podaja Slika 7.12. Zanjo potrebujemo štiri stanja, kar je posledica komunikacije s pomnilniškim vmesnikom.

Vsaka od štirih inštrukcij ima svojo sekvenco stanj ali detajl DPS, s katerim KA krmili izvršilno enoto ozr. njene resurse. Prikazujejo jih po vrsti Slika 7.13 (Load), Slika 7.14 (Store), Slika 7.15 (Add) in Slika 7.16 (Branch).

Pri združevanju zgornjih segmentov na osnovi visokonivojskega DPS (glej Sliko 7.11) pa je mogoče izvesti določene optimizacije, eliminacije posameznih stanj, kar je posledica stikov med posameznimi detajlnimi segmenti DPS.



Slika 7.12: Detajl resetiranja in dostave inštrukcije.

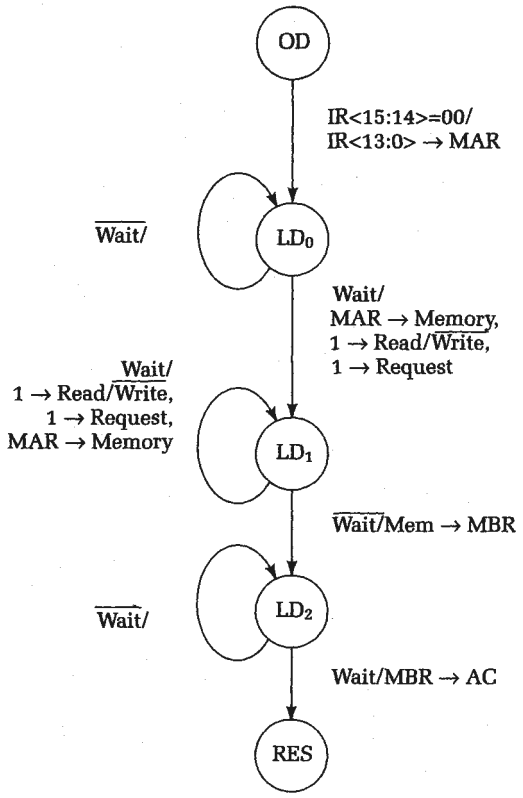
Tako je mogoče izpustiti stanje ST_2 (ki se nahaja v sekvenci izvršitve Store inštrukcije), saj dostava že testira ali je pomnilnik pripravljen sprejeti novo zahtevo s testiranjem, ali je Wait signal aktiven (enak 1) ali ne. Iz istega razloga je mogoče izpustiti zanko $\overline{\text{WAIT}}$ in izhod WAIT pri stanjih LD_2 in AD_2. Podobno lahko eliminiramo isto zanko in izhod iz stanj LD_0, ST_0 in AD_0, saj že IF_2 kompletira protokol med procesorjem in pomnilnikom. Tako dobimo ob upoštevanju omenjenih vsebinskih poenostavitev končni DPS, s katerim KA krmili izvajanje operacij preprostega procesorja. Prikazuje ga Slika 7.17.

Na zgornji sliki zaradi preglednosti niso podane izhodne (Mealyjeve) spremenljivke, zato jih podajamo spodaj po pomenu:

Krmilni signal in vhodni pogoji:

Reset

WAIT



Slika 7.13: Izvršilna sekvenca inštrukcije 'Load'.

IR<15:14>

AC<15>

Krmilni izhodni signali:

0 → PC

PC+1 → PC

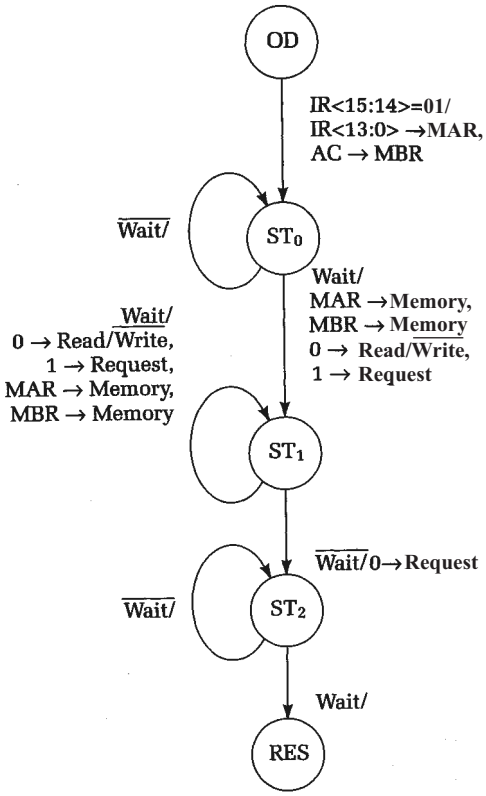
PC → MAR

MAR → Memory Address Bus

Memory Data Bus → MBR

MBR → Memory Data Bus

MBR → IR



Slika 7.14: Izvršilna sekvenca inštrukcije 'Store'.

MBR → AC

AC → MBR

AC+MBR → AC

IR<13:0> → MAR

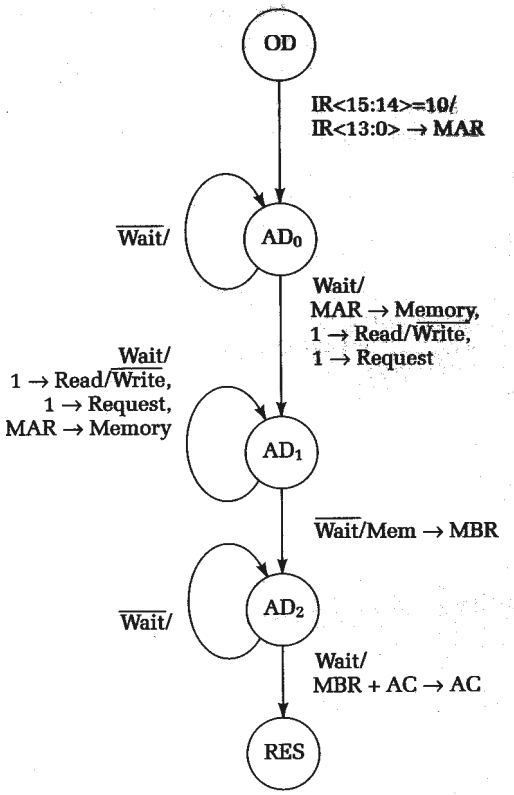
IR<13:0> → PC

1 → READ/$\overline{\text{WRITE}}$

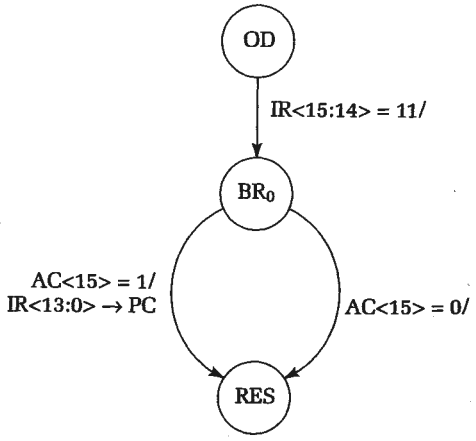
0 → READ/$\overline{\text{WRITE}}$

1 → REQ

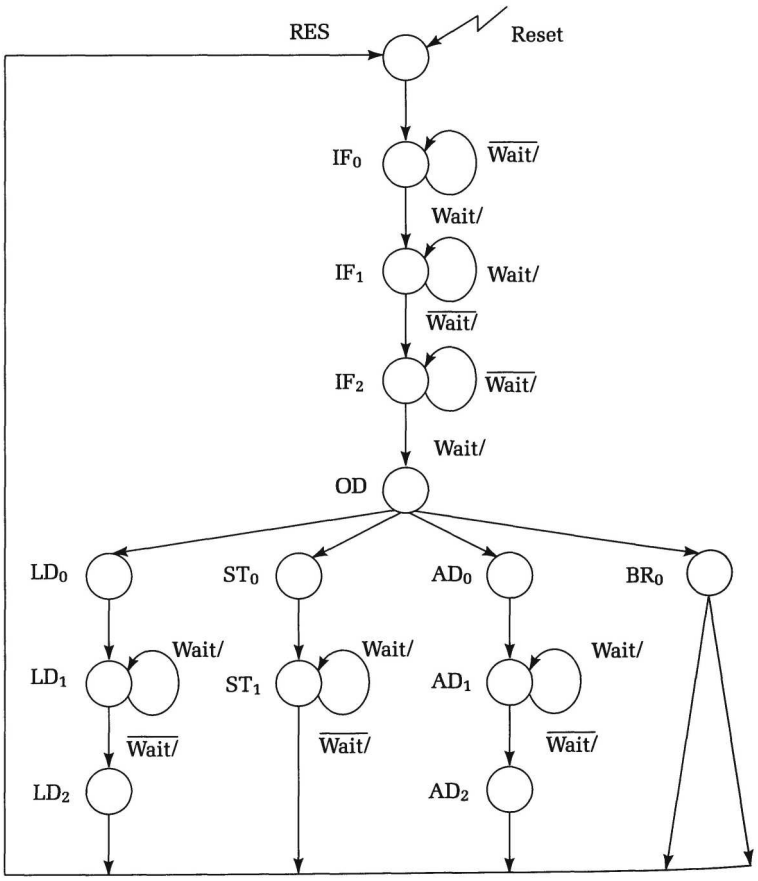
Glede na operacije krmiljena izvršilne enote s strani KA je potrebno povedati še naslednje.



Slika 7.15: Detajl DPS za inštrukcijo 'Add'.



Slika 7.16: Segment DPS inštrukcije 'Branch'.



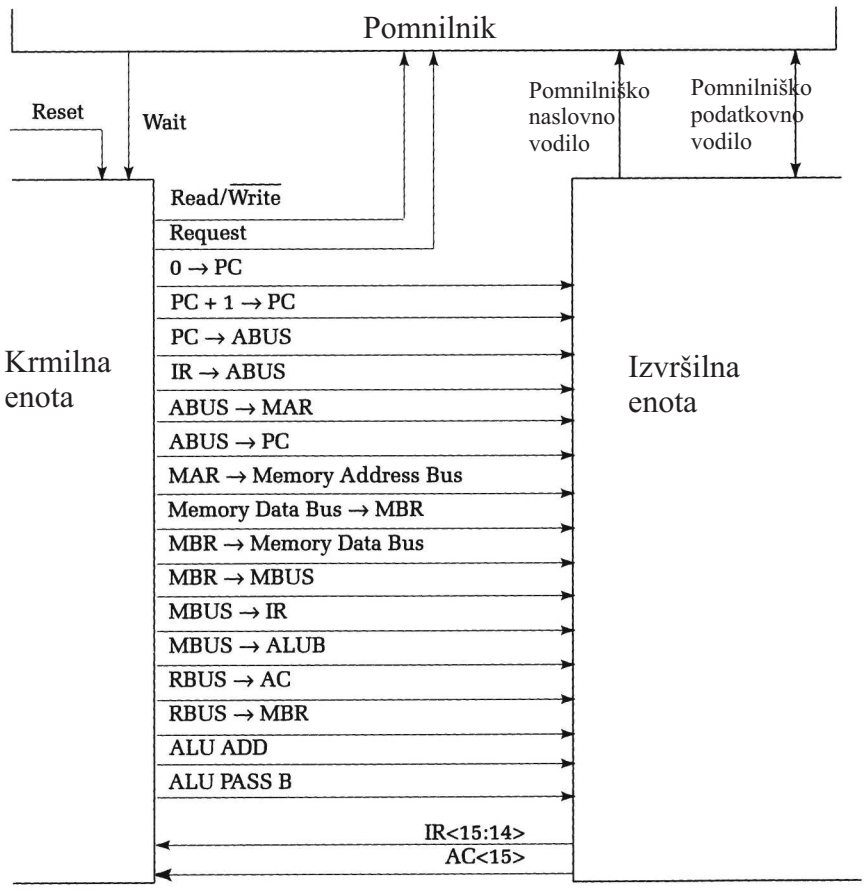
Slika 7.17: Kompletni diagram prehajanja stanj KA tipa Mealy za preprost procesor.

Prvič, nekateri krmilni signali izvedejo takojšnjo realizacijo, drugi pa so zakasnjeni do naslednjega urinega signala. V primeru, ko je ponorni resurs vodilo, se krmiljenje izvede takoj, če pa je ponor pomnilni element (n.pr. register), pa se krmilni signal realizira zakasnjeno.

In drugič, v jeziku RTN smo podajali prenose podatkov med resursi, ki pa se ne izvedejo vso v enem časovnem ciklu. Zato govorimo o mikrooperacijah, ki pa se dejansko izvedejo ob enemu urinemu signalu. Zato je pomembno vedeti, katere mikrooperacije zahteva posamezni RTN zapis, saj tako podamo tudi časovno dimenzijo izvrševanja prenosov. V naslednji tabeli podajamo povezavo RTN prenosov in potrebne mikrooperacije zanje:

RTN prenos:	**Mikrooperacije:**
0 → PC	0 → PC (zakasnjeno);
PC+1 → PC	PC+1 → PC (zakasnjeno);
PC → MAR	PC → ABUS (takoj), ABUS → MAR (zakasnjeno);
MAR → Address Bus	MAR → Address Bus (takoj);
Data Bus → MBR	Data Bus → MBR (zakasnjeno);
MBR → Data Bus	MBR → Data Bus (takoj);
MBR → IR	MBR → MBUS (takoj), MBUS → IR (zakasnjeno);
MBR → AC	MBR → MBUS (takoj), MBUS → ALU B (takoj), ALU PASS B (takoj), ALU result → RBUS (takoj), RBUS → AC (zakasnjeno);
AC → MBR	AC → ALU A (takoj), ALU PASS A (takoj), ALU Result → RBUS (takoj), RBUS → AC (zakasnjeno);
AC+MBR → AC	AC → ALU A (takoj), MBR → MBUS (takoj), MBUS → ALU B (takoj), ALU ADD (takoj), ALU Result → RBUS (takoj), RBUS → AC (zakasnjeno);
IR<13:0> → MAR	IR → ABUS (takoj), ABUS → IR (zakasnjeno);
IR<13:0> → PC	IR → ABUS (takoj), ABUS → PC (zakasnjeno);
1 → READ/$\overline{\text{WRITE}}$	READ (takoj);
0 → READ/$\overline{\text{WRITE}}$	WRITE (takoj);
1 → REQ	REQUEST (takoj);

Končno podajmo še prikaz mikrooperacij ozr. njihove izvore in ponore za slučaj preprostega procesorja. Prikazuje ga Slika 7.18.



Slika 7.18: Prikaz mikrooperacij preprostega procesorja.

Poglavje 8

IMPLEMENTACIJA KRMILNIKA CENTRALNE PROCESNE ENOTE

V tem poglavju si bomo ogledali različne možne implementacije krmilnega dela CPE. Najprej bomo podali klasično metodo realizacijo KA tipa Mealy ozr. Moore z naključno logiko (angl. random logic), kjer bodo namesto SSI (angl. Small Scale Integration) operatorjev, uporabljeni ROM pomnilni (angl. Read Only Memory) elementi.

Druga metoda je dekompozicijska, ker razdeli DPS KA v več enostavnejših DPS, ki med seboj komunicirajo preko vhodnih/izhodnih spremenljivk. Obstaja strategija za delitev KA, ki se dobro ujema s strukturo krmilnika procesorja.

Tretja metoda uporablja števce za realizacijo prehodov med stanji. Ta pristop uporablja MSI (angl. Medium Scaale Integration) kot so števci, multipleksorji (MX) in dekodirniki (glej poglavji 3 in 5).

Zadnja metoda se imenuje mikroprogramiranje in uporablja ROM elemente za kodiranje naslednjih stanj in krmilnih ukazov direktno z bitnimi zapisi v pomnilniku. Pri tem bodo opisane tri alternativne variante mikroprogramiranja: s skočnim sekvenčnikom, horizontalnim in vertikalnim mikroprogramiranjem. Prva kodira številna možna naslednja stanja v okviru ROM elementa, druga

dodeljuje po en bit v ROM-u za vsak izhod krmilnika, tretja pa racionalno kodira krmilne izhode in s tem reducira širino ROM pomnilnika.

8.1 Naključna logika

V tem primeru so funkcije prehajanja stanj in izhodne funkcije formulirane z izrazi diskretne logike, ki ustrezajo logičnim vratom (AND, OR, NAND, NOR). Pri realizaciji takšnih izrazov pa lahko enostavno uporabljamo LSI programabilno logiko, n.pr. PAL, PLA, PLE, CPLD, FPGA.

Najprej bomo za primer preprostega procesorja iz prejšnjega poglavja, podali Moorovo realizacijo, nato pa še primerjali sinhronsko in asinhronsko Mealyjevo realizacijo, saj je običajno implementacija Mealyjevega stroja običajno ekonomičnejša od Moorove.

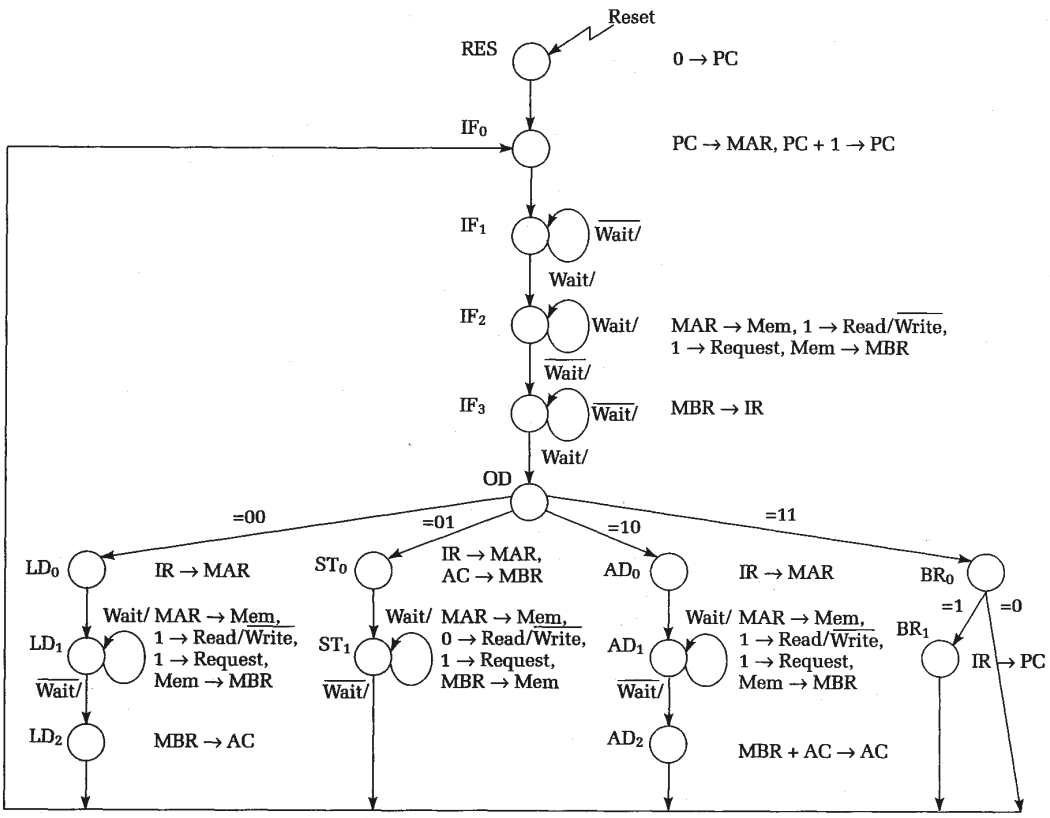
8.1.1 Moorov stroj

Kompleten DPS za Moorov stroj je za slučaj preprostega procesorja podan na Sliki 8.1.

Kot je razvidno iz primerjave Mealyjevega in Moorovega stroja, ima slednji več stanj, kar je značilnost teh strojev, čeprav je zaradi enostavnega primera razlika majhna. Dodatno stanje je potrebno pri dostavi 'Reset' inštrukcije in eno pri skočni sekvenci.

Povezovanje operacij s stanji stroja je precej enoumno. Samo ena kombinacija operacij pri istem stanju je presenetljiva in to je pri stanjih IF_2, LD_1in AD_1, kjer se hkrati izvede zahteva po čitanju iz pomnilnika in vpis iz pomnilniškega podatkovnega vodila v MBR. Vendar podrobna analiza časovnega diagrama, podaja jo Slika 8.2, pokaže, da operaciji korektno izvršita svoji nalogi.

Blok diagram Moorovega stroja prikazuje Slika 8.3. Ker ima DPS 16 stanj, potrebujemo 4 FF za register stanj. Logika za izračun naslednjega stanja ima 9 vhodov (4 od trenutnega stanja in vhodi Reset, WAIT, dva IR bita in en AC bit) in 4 izhode (naslednje stanje). Izhodna logika pa ima 4 vhode (trenutno stanje) in 18 krmilnih izhodov.

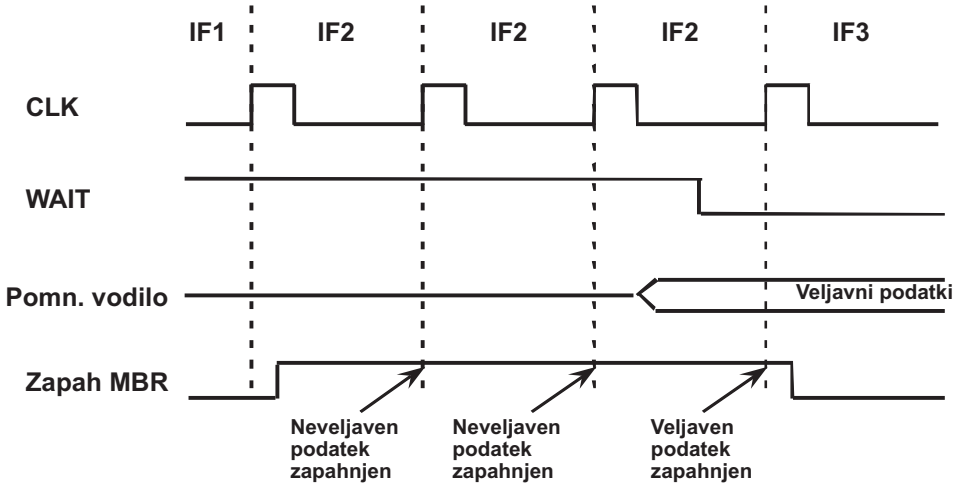


Slika 8.1: Kompleten diagram prehajanja stanj Moorovega stroja.

Če uporabimo ROM, lahko logiko za naslednje stanje implementiramo z elementom 512 x 4 ter izhodno logiko z ROM elementom velikosti 16 x 18. Aplikacijsko tabelo za obe logiki, za naslednje stanje in izhodno logiko, podaja Slika 8.4.

Aplikacijska tabela ponuja naslednja opažanja. Prvič, levi del tabele vsebuje precejšno število redundanc (angl. don't care) in drugič, število prenosov na stanje je majhno (največ 4). Nekaj izhodnih signalov se tudi vedno pojavlja skupaj.

Z ROM elementi seveda ne moremo izkoristiti redundance, vendar pa kodiranje stanj ni kritično. N.pr. z uporabo PAL ali PLA vezij in dobrim kodiranjem stanj lahko znatno zmanjšamo kompleksnost logike za naslednje stanje.



Slika 8.2: Detajl časovnega diagrama za Moorov stroj.

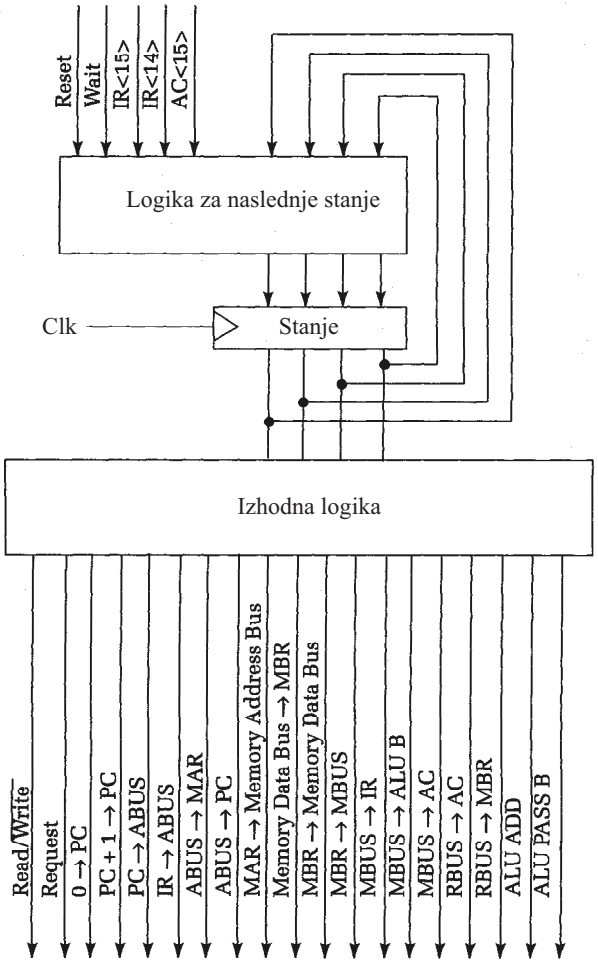
8.1.2 Sinhronski Mealyjev stroj

Takšen stroj se ne razlikuje močno od pravkar opisanega Moorovega stroja. Potrebno je le povezati funkcije za naslednje stanje in izhodne funkcije v en sam logični blok. Pri Mealyjevem stroju ima takšen logični blok 9 vhodov in 22 izhodov. Če takšen skupni logičen blok realiziramo z ROM elementom, pridemo do zanimivih primerjav.

V primeru Moorovega stroja smo potrebovali ROM velikosti 2336 bitov = 512 x 4 + 16 x 18, pri Mealyjevem stroju pa ROM velikosti 11264 bitov = 512 x 22. To seveda kaže na neučinkovitost realizacije z ROM elementom. Vendar bomo pri mikroprogramskih variantah spoznali precej boljši izkoristek ROM elementov.

Sinhronizacijo Mealyjevega stroja, ki ima tudi izhode sinhronizirane, lahko dosežemo na tri načine. Prikazuje jih Slika 8.5. Uporabimo lahko frontno prožene FF a. na vhodu in izhodu, b. samo na vhodu in c. samo na izhodu.

Časovni diagrami vseh treh variant so podani na Sliki 8.6. Kot je razvidno, sta varianti b. in c. boljši, saj zahtevata le dve stanji, medtem ko je varianta a. dražja in potratnejša glede na število stanj in FF ter posledično tudi počasnejša.



Slika 8.3: Blok diagram Moorovega stroja.

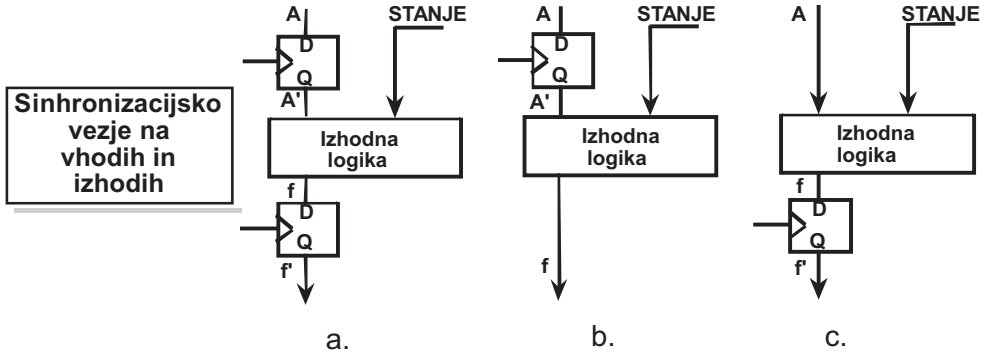
8.2 Dekompozicija stanj (deli in vladaj)

Klasično implementacijo KA, ki smo si jo ogledali v prejšnjem poglavju, imenujemo tudi monolitno, kar pa je zaradi obsežnega KA lahko velik problem. Alternativno rešitev predstavlja pristop, ki KA najprej razdeli na manjše dele in nato vsakega posebej realizira.

Običajno se KA krmilne enote CPU razdeli na tri manjše KA*: na časovni KA*, inštrukcijski KA* in pogojni KA*.

Reset	Wait	IR<15>	IR<14>	AC<15>	Trenutno stanje	Naslednje stanje	Register Transfer operacije
1	X	X	X	X	X	RES (0000)	0 →PC
0	X	X	X	X	IF0 (0001)	IF1 (0001)	PC→ MAR, PC + 1 → PC
0	0	X	X	X	IF1 (0010)	IF1 (0010)	
0	1	X	X	X	IF1 (0010)	IF2 (0011)	
0	1	X	X	X	IF2 (0011)	IF2 (0011)	MAR → Mem, Read,
0	0	X	X	X	IF2 (0011)	IF3 (0100)	Request, Mem → MBR
0	0	X	X	X	IF3 (0100)	IF3 (0100)	MBR → IR
0	1	X	X	X	IF3 (0100)	OD (0101)	
0	X	0	0	X	OD (0101)	LD0 (0110)	
0	X	0	1	X	OD (0101)	ST0 (1001)	
0	X	1	0	X	OD (0101)	AD0 (1011)	
0	X	1	1	X	OD (0101)	BR0 (1110)	
0	X	X	X	X	LD0 (0110)	LD1 (0111)	IR → MAR
0	1	X	X	X	LD1 (0111)	LD1 (0111)	MAR → Mem, Read,
0	0	X	X	X	LD1 (0111)	LD2 (1000)	Request, Mem → MBR
0	X	X	X	X	LD2 (1000)	IF0 (0001)	MBR → AC
0	X	X	X	X	ST0 (1001)	ST1 (1010)	IR→ MAR, AC → MBR
0	1	X	X	X	ST1 (1010)	ST1 (1010)	MAR → Mem, Write,
0	0	X	X	X	ST1 (1010)	IF0 (0001)	Request, MBR → Mem
0	X	X	X	X	AD0 (1011)	AD1 (1100)	IR →MAR
0	1	X	X	X	AD1 (1100)	AD1 (1100)	MAR → Mem, Read,
0	0	X	X	X	AD1 (1100)	AD2 (1101)	Request, Mem → MBR
0	X	X	X	X	AD2 (1101)	IF0 (0001)	MBR + AC → AC
0	X	X	X	0	BR0 (1110)	IF0 (0001)	
0	X	X	X	1	BR0 (1110)	BR1 (1111)	
0	X	X	X	X	BR1 (1111)	IF0 (0001)	IR →PC

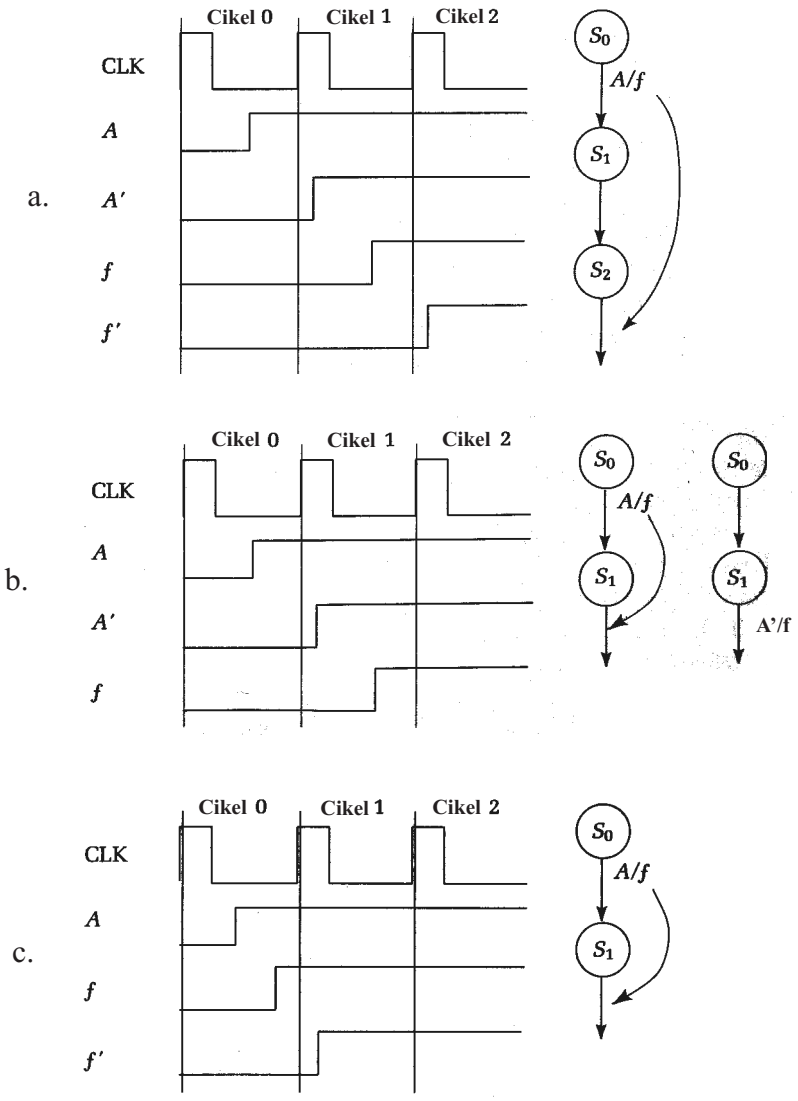
Slika 8.4: Simbolična aplikacijska tabela za Moorov stroj.



Slika 8.5: Variante sinhronizacije Mealyjevih izhodov.

Časovni KA* določa faze izvajanja inštrukcij, n.pr. dostava, dekodiranje, eksekucija, pri čemer so omenjena stanja enaka za vse inštrukcije. Ta KA* opisuje torej značilne faze izvajanja inštrukcij, ne glede na specifične posebnosti posameznih imštrukcij.

Inštrukcijski KA* identificira trenutno inštrukcijo, kar pomeni, da njegovo



Slika 8.6: Časovni diagrami variant za sinhronizacijo Mealyjevih izhodov (a,b,c).

stanje sledi inštrukcijskemu dekodiranju.

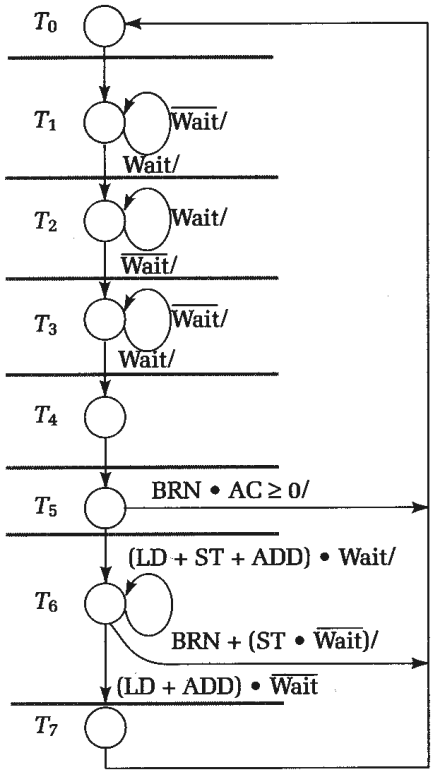
Pogojni KA* predstavlja stanje pogoja iuzvršilne enote, ki je predmet testiranja določenih skočnih inštrukcij.

Prednost delitve KA na omenjene tri KA* ima prednost, ker je prehod na

ustrezne DPS* zelo enostaven in izhaja neposredno iz DPS KA.

8.2.1 Parcialni KA* za slučaj preprostega procesorja

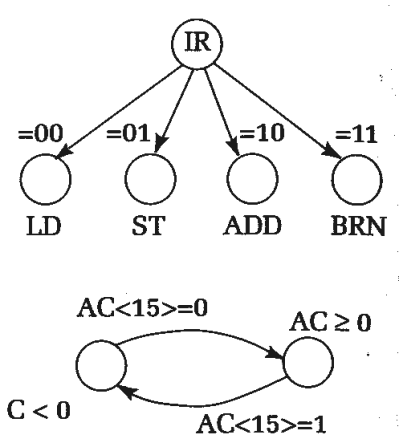
Kot izhodišče nam služi DPS za KA na Sliki 8.1. Da iz njega dobimo DPS* za časovni KA*, moramo poiskati najslabšo (najdaljšo) pot v DPS. Za slučaj preprostega procesorja ozr. DPS s Slike 8.1 ima najdaljša pot 8 stanj (pri inštrukcijah ADD in LOAD). Sedaj je potrebno dobljeno osem-stanjsko sekvenco parametrizirati z izhodi različnih inštrukcij (v primeru preprostega procesorja LD, ST, ADD, BRN) in pogojnih stanj (v našem primeru najvišji bit AC, ki določa predznak števila: AC $\geq$ 0 (+), AC $<$ 0 (-)). Te izhode povežemo s prehodi v DPS* tako, kot to prikazuje Slika 8.7.



Slika 8.7: Diagram prehajanja stanj (DPS*) časovnega avtomata (KA*).

Na zgornji sliki je izpuščeno stanje Reset, ki ga uporablja drugi KA*. Inštrukcijski

KA* in pogojni KA* ozr. njuna DPS* pa sta prikazana na Sliki 8.8.



Slika 8.8: DPS* za inštrukcijki KA* in pogojni KA*.

Vsaka inštrukcija, ne glede na tip, sledi sekvenci prvih 5 stanj, od IF_0 preko IF_3 do OD. Ta stanja so v časovnem DPS* označena z T_0 do T_4.

Nato sledi razcep v originalnem DPS, glede na tip inštrukcije. Kljub temu je v časovnem DPS* nekaj stanj enakih pri različnih inštrukcijah. Stanja LD_0 - LD_2, ST_0 - ST_1, AD_0 - AD_2, BR_0 - BR_1 se reducirajo v stanja T_5, T_6, in T_7. Edini izhod iz časovnega KA* je trenutno stanje, od T_0 do T_7.

Na Sliki 8.7 je prikazano, kako izhodi iz inštrukcijskega in pogojnega DPS* vplivajo na prehajanje stanj časovnega KA*. N.pr. v stanju T_5 in slučaju, ko je AC ≥ 0, se časovni KA* vrne v stanje T_0, sicer gre naprej v T_6. Če je časovni KA* v stanju T_6 in če se izvaja ena od inštrukcij LD, ST ali ADD ter je aktiven signal WAIT, potem časovni KA* ostane v T_6. Če pa se izvaja inštrukcija BRN ali ST in je WAIT resetiran, se vrne časovni KA* v T_0, sicer nadaljuje v T_7.

8.2.2 Generiranje mikrooperacij

Na osnovi parcialnih DPS sedaj ni težko generirati mikrooperacij. Različne registrske prenosne operacije iz RTN zapisa se izvedejo ob naslednjih pogojih:

RTN zapis:	**Pogoj:**
0 → PC:	Reset
PC+1 → PC:	T_0
PC → MAR:	T_0
MAR → Memory Address Bus:	T_2 V T_6 . (LD ∨ ST ∨ ADD)
Memory Data Bus: → MBR:	T_2 V T_6 . (LD ∨ ADD)
MBR → Memory Data Bus:	T_6 . ST
MBR → IR:	T_4
MBR → AC:	T_7 . LD
AC → MBR:	T_5 . ST
AC + MBR → AC:	T_7 . ADD
IR<13:0> → MAR:	T_5 . (LD ∨ ST ∨ ADD)
IR<13:0> → PC:	T_6 . BRN
1 → READ/$\overline{\text{WRITE}}$:	T_2 V T_6 . (LD ∨ ADD)
0 → READ/$\overline{\text{WRITE}}$:	T_6 . ST
1 → REQ:	T_2 V T_6 . (LD ∨ ST ∨ ADD)

Kot je razvidno, stanja pogojnega KA* niso eksplicitno uporabljena pri generiranju registrskih prenosov, seveda pa vplivajo na prehajanje stanj časovnega KA*.

8.3 Skočni števci

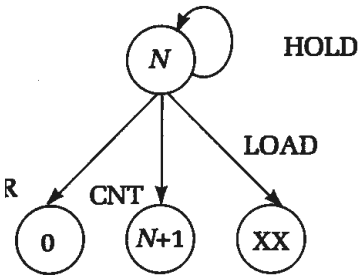
V tem primeru uporabljamo operatorje MSI za realizacijo KA. Pri tem nam števni register služi kot register stanj KA. Najprej si bomo ogledali osnovno ali 'čisto' varianto realizacije KA s števnim registrom, nato pa še hibridno izpeljanko.

8.3.1 Čisti skočni števec

Čisti skočni števec (angl. pure jump counter) omogoča samo štiri možna naslednja stanja: ohranitev obstoječega stanja (ukaz 'hold' za števec), skok na začetno stanje 0 (ukaz 'clear'), naslednje stanje (ukaz ' count' ozr. povečaj

števec za 1) in skok na novo stanje (ukaz 'load'). Pri čistem skočnem števcu je naslednje stanje izključno funkcija trenutnega stanja.

Slika 8.9 prikazuje možnosti za naslednje stanje, ki jih omogoča čisti skočni števec.



Slika 8.9: Možnosti naslednjega stanja pri čistem skočnem števcu.

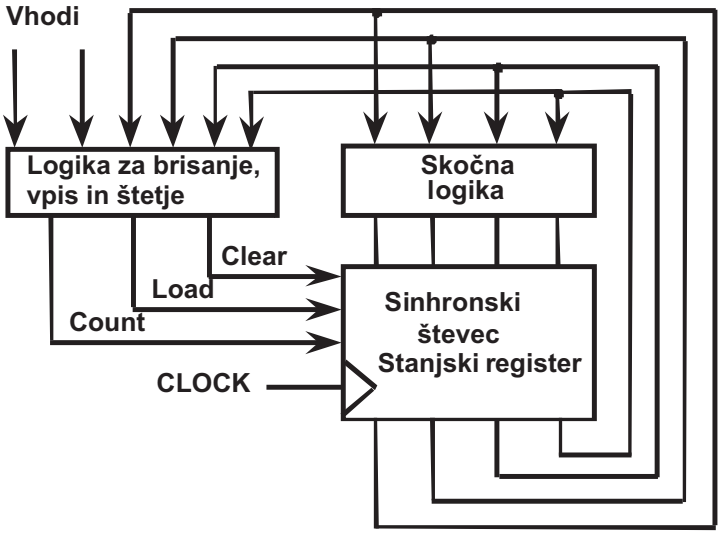
Blok shemo čistega skočnega števca pa podaja Slika 8.10. Na sliki vidimo poleg števnega registra še dva logična bloka, prvi v odvisnosti od stanja in vhodov določa ukaze 'clear', 'load' in 'count' za števec, drugi pa na osnovi stanja pripravi adreso novega stanja, ki se ob ukazu 'load' vpiše v števni register. Oba logična bloka je mogoče realizirati z diskretnimi vrati (SSI), ali programabilnimi vezji PAL, PLA ali PLE (LSI).

V primeru, ko imamo več možnih naslednjih stanj kot štiri, moramo ustrezni DPS spremeniti tako, da bo ob vsakem stanju število naslednjih stanj manjše ali enako 3. To zahteva uporabo novih stanj, kar podaljša in podraži realizacijo. Slika 8.11 prikazuje spremembo DPS s 4 stanji v naslednjem koraku in nov DPS z dvema stanjema več.

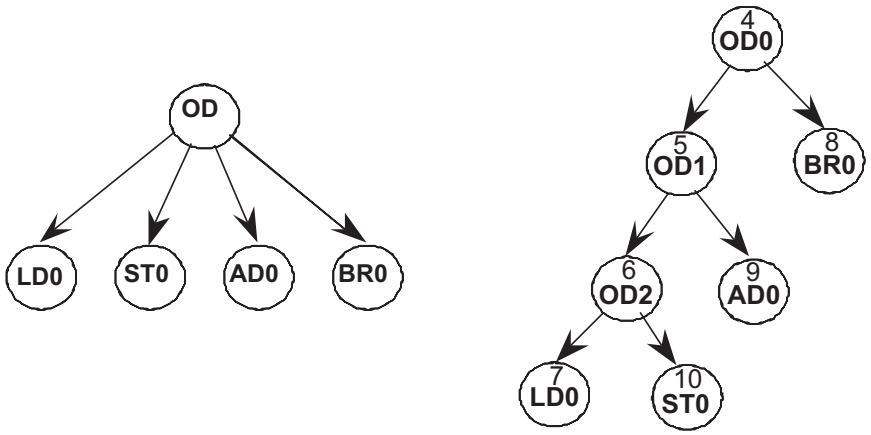
8.3.2 Hibridni skočni števec

S hibridnim skočnim števcem rešujemo probleme večjega števila stanj v naslednjem koraku. V tem primeru imamo namesto dveh en sam logični blok, poleg števnega registra, kar ilustrira Slika 8.12.

Oglejmo si implementacijo Mealyjevega KA s Slike 7.17 s hibridnim skočnim števcem. Najprej moramo rešiti problem kodiranja stanj. Kodiranje, ki upošteva



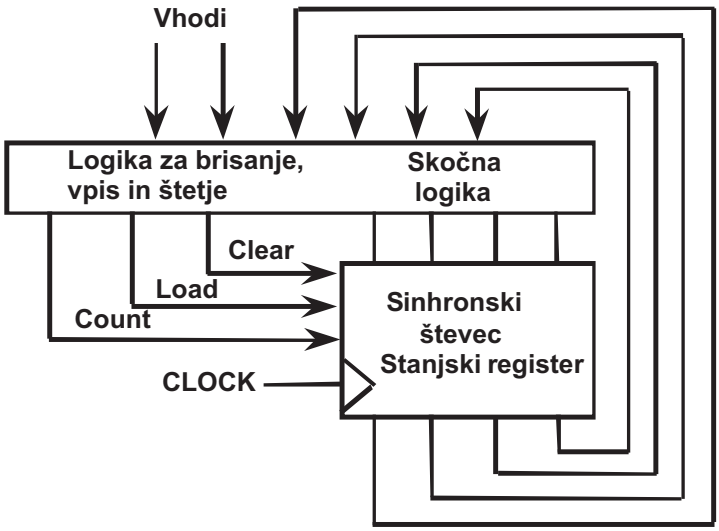
Slika 8.10: Blok shema čistega skočnega števca.



Slika 8.11: Modifikacija DPS v primeru prevelikega števila naslednjih stanj.

vrstni red v DPS, torej po vrsti od začetka proti koncu označuje stanja, je podano na Sliki 8.13.

Stanje RES (Reset) kodiramo s kodo 0, ker se najbolj pogosto pojavlja. Nato za vsako naslednje stanje povečamo kodo za 1, inštrukcije pa obravnavamo po vrsti od leve proti desni. Sedaj identificiramo potencialne skoke v posameznih stanjih. Vidimo, da ima le stanje OD zahtevo po štirih naslednjih stanjih, kar presega zmožnosti čistega skočnega števca. Vendar je mogoče naslednja stanja



Slika 8.12: Blok shema hibridnega skočnega števca.

v tem primeru opisati z trenutnim stanjem in operacijsko kodo inštrukcije, ki se nahaja v IR registru. Zato je logika za določitev skočnega stanja odvisna le od IR<op code> bitov. Kot je razvidno iz Slike 8.13, so stanja označena s kodami od 0 do 13.

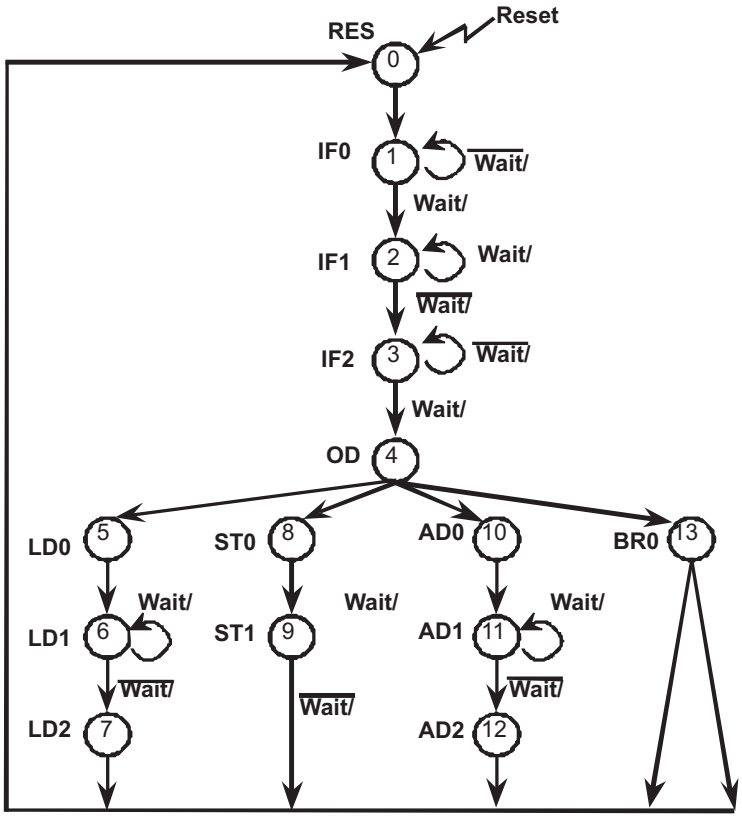
Za prenose v stanje RES (0), mora biti aktiven signal 'clear', ki števec postavi na 0 in s tem realizira prehod v stanje RES (0). Prehod v RES stanje se pojavlja v stanjih S_7, S_9 (in $\overline{\text{WAIT}}$), S_{12} in S_{13}.

Signal 'load' za števec se aktivira samo v primeru stanj z večizhodnimi skoki. V našem primeru je to stanje S_4 (OD). IR bita (op code) določata novo stanje, ki se vpiše v števni register.

Aktiviranje 'count' signala za štetje ozr. povečanje kode stanja za 1, je nekoliko bolj zapleteno. Zato za preostala signala, t.j. 'count' in 'hold' podajamo preklopni funkciji, ki ustrezno aktiviranje signalov pogojujeta:

$$\begin{aligned} CNT &= (S_0 \vee S_5 \vee S_8 \vee S_{10}) \vee \text{WAIT} \cdot (S_1 \vee S_3) \vee \overline{\text{WAIT}} \cdot (S_2 \vee S_6 \vee S_9 \vee S_{11}) \\ HOLD &= \overline{\text{WAIT}} \cdot (S_1 \vee S_3) \vee \text{WAIT} \cdot (S_2 \vee S_6 \vee S_9 \vee S_{11}) \end{aligned}$$

Pri tem velja opozoriti, da je HOLD enačba ($\overline{\text{CNT}}$) enostavnejša od CNT



Slika 8.13: Kodiranje po vrsti (angl depth-first) za Mealyjev KA.

enačbe. Zato se izplača, da implementiraamo HOLD in jo nato invertiramo, da dobimo CNT signal.

Skočna logika v stanju S_4 je enostavna. Zanjo lahko uporabimo 4 x 4 ROM, za katerega sta adresni liniji kar 'op code' bita iz IR registra.

8.3.3 Generiranje mikrooperacij

Mikrooperacije lahko generiramo iz dekodiranih stanj in vhodov. Ustrezni logični izrazi so podani spodaj. Za sinhronizacijo operacij moramo vhodna signala WAIT in RESET sinhronizirati preden ju uporabimo.

RTN zapis: **Pogoj:**

$0 \rightarrow$ PC:	Reset
PC + 1 $\rightarrow$ PC:	S_0
PC $\rightarrow$ MAR:	S_0
MAR $\rightarrow$ Memory Address Bus:	$\text{WAIT} \cdot (S_1 \vee S_2 \vee S_5 \vee S_6 \vee S_8 \vee S_9 \vee S_{11} \vee S_{12})$
Memory Data Bus $\rightarrow$ MBR:	$\overline{\text{WAIT}} \cdot (S_2 \vee S_6 \vee S_{11})$
MBR $\rightarrow$ Memory Data Bus:	$\text{WAIT} \cdot (S_8 \vee S_9)$
MBR $\rightarrow$ IR:	$\text{WAIT} \cdot S_3$
MBR $\rightarrow$ AC:	$\text{WAIT} \cdot S_7$
AC $\rightarrow$ MBR:	$IR_{15} \cdot IR_{14} \cdot S_4$
AC + MBR $\rightarrow$ AC:	$\text{WAIT} \cdot S_{12}$
IR<13:0> $\rightarrow$ MAR:	$(\overline{IR_{15}} \cdot \overline{IR_{14}} \vee \overline{IR_{15}} \cdot IR_{14} \vee IR_{15} \cdot \overline{IR_{14}}) \cdot S_4$
IR<13:0> $\rightarrow$ PC:	$AC_{15} \cdot S_{13}$
1 $\rightarrow$ READ/$\overline{\text{WAIT}}$:	$\text{WAIT} \cdot (S_1 \vee S_2 \vee S_5 \vee S_6 \vee S_{11} \vee S_{12})$
0 $\rightarrow$ READ/$\overline{\text{WAIT}}$:	$\text{WAIT} \cdot (S_8 \vee S_9)$
1 $\rightarrow$ REQ:	$\text{WAIT} \cdot (S_1 \vee S_2 \vee S_5 \vee S_6 \vee S_8 \vee S_9 \vee S_{11} \vee S_{12})$

Shematični opis realizacije KA s skočnim števcem podaja Slika 8.14. Glavne komponente so: sinhronski števec (74163), dekodirnik 4/16 (74154), skočna logika 4 x 4 ROM), PAL za realizacijo 'count' funkcije, diskretna logika za realizacijo 'clear' funkcije.

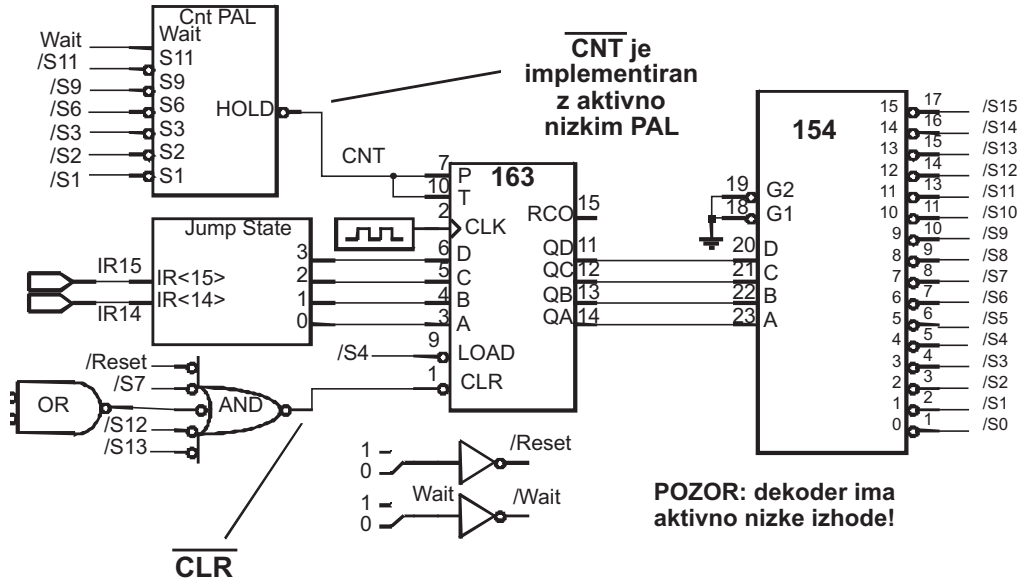
Realizacija KA s skočnimi števci je smiselna za slučaje, ko je število stanj v območju od 16 do 32.

Splošna strategija zamenjave zunanje logike, kot so vrata, MX, PAL z ROM se imenuje mikroprogramiranje. Takšen način realizacije KA bomo spoznali v naslednjem poglavju.

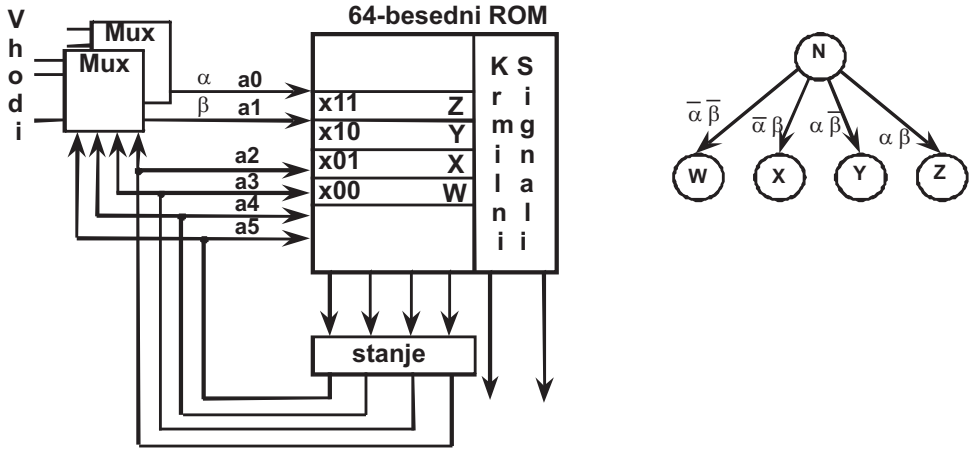
8.4 Skočni sekvenčniki

Najprej poglejmo, kako je mogoče z ROM realizirati sekvenčnik, ki omogoča do štiri razvejitve na stanje, ozr. kjer je število naslednjih stanj navzgor omejeno s 4. Za slučaj Mealyjevega stroja je takšen sekvenčnik prikazan na Sliki 8.15.

V ROM elementu zgornji 4 adresni biti določajo stanje, spodnja dva bita pa sta vhoda α in β, ki sta odvisna od stanja in vhodov. Realizirana sta z dvema



Slika 8.14: Shematski opis realizacije KA s skočnim števcem.



Slika 8.15: Blok diagram sekvenčnika s štirimi razvejitvami na stanje.

MX, ki ju naslavljajo biti stanja, podatkovne vhode pa določajo vhodi v KA. ROM vsebuje v vsaki vrstici 4 bite za naslednje stanje in krmilne signale za nadzor nad izvršilno enoto ozr. prenosi med registri.

Vzemimo kot primer Mealyjev KA za preprost procesor iz Slike 7.17. Tedaj so vhodi v KA le štirje: WAIT, AC15, IR15 in IR14. V OD stanju ('Operation Decode') sta pomembna samo bita iz IR za določitev naslednjega stanja. V

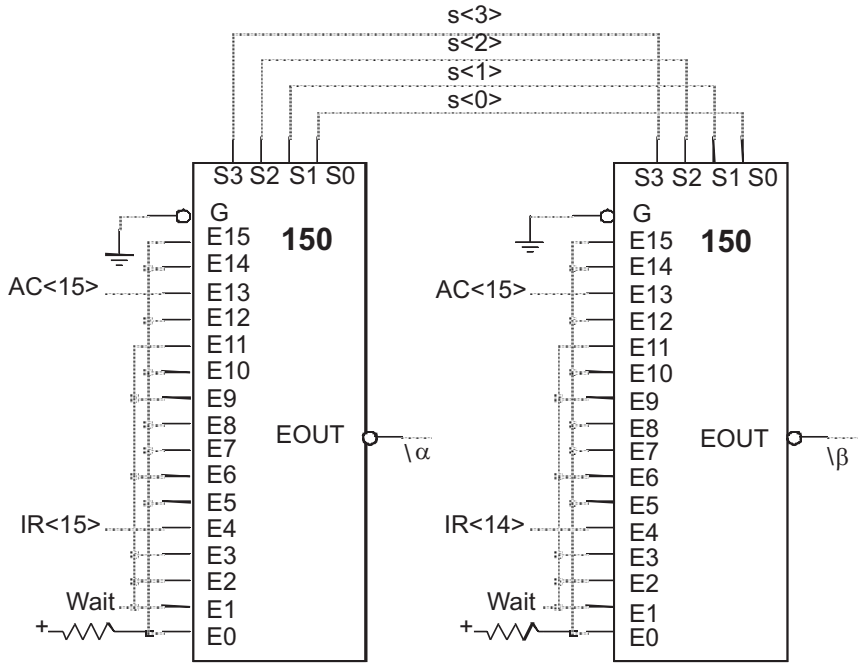
stanju, ki izvršuje BRN inštrukcijo pa KA upošteva le AC15 bit, da določi novo stanje. Mnogo drugih stanj v ta namen potrebuje le WAIT signal. Slika 8.16 prikazuje vsebino ROM za slučaj realizacije Mealyjevega KA z ROM. Adreso določajo RESET, trenutno stanje (4 biti) in pogojna vhoda α in β. Z dodanim RESET signalom smo podvojili vsebino ROM, kar pa je najenostavnejši način za realizacijo 'Reset' funkcije.

	ROM naslov			Vsebina ROMa	
	(Reset,	Trenutno stanje,	α, β)	Naslednje stanje	Register Transfer operacije
RES	0	0000	X X	0001 (IF0)	PC → MAR, PC + 1 → PC
IF0	0	0001	0 0	0001 (IF0)	
	0	0001	1 1	0010 (IF1)	MAR → Mem, Read, Request
IF1	0	0010	0 0	0011 (IF2)	MAR → Mem, Read, Request
	0	0010	1 1	0010 (IF1)	Mem → MBR
IF2	0	0011	0 0	0011 (IF2)	
	0	0011	1 1	0100 (OD)	MBR → IR
OD	0	0100	0 0	0101 (LD0)	IR → MAR
	0	0100	0 1	1000 (ST0)	IR → MAR, AC → MBR
	0	0100	1 0	1001 (AD0)	IR → MAR
	0	0100	1 1	1101 (BR0)	IR → MAR
LD0	0	0101	X X	0110 (LD1)	MAR → Mem, Read, Request
LD1	0	0110	0 0	0111 (LD2)	Mem → MBR
	0	0110	1 1	0110 (LD1)	MAR → Mem, Read, Request
LD2	0	0111	X X	0000 (RES)	MBR → AC
ST0	0	1000	X X	1001 (ST1)	MAR → Mem, $\overline{\text{Write}}$, Request, MBR → Mem
ST1	0	1001	0 0	0000 (RES)	
	0	1001	1 1	1001 (ST1)	MAR → Mem, $\overline{\text{Write}}$, Request, MBR → Mem
AD0	0	1010	X X	1011 (AD1)	MAR → Mem, Read, Request
AD1	0	1011	0 0	1100 (AD2)	
	0	1011	1 1	1011 (AD1)	MAR → Mem, Read, Request
AD2	0	1100	X X	0000 (RES)	MBR + AC → AC
BR0	0	1101	0 0	0000 (RES)	
	0	1101	1 1	0000 (RES)	IR → PC
	1	XXXX	X X	0000 (RES)	PC → 0

Slika 8.16: Izvedba Mealyjevega KA z ROM.

Za realizacijo α in β funkcij lahko uporabimo dva MX (Slika 8.17). IR15 in IR14 sta povezana na istoležna vhoda obeh MX, ki ju naslavlja stanje OD. WAIT je pripeljan na vhode 1,2,3,6,9 in 11 obeh MX, ki jih naslavljajo stanja, v katerih je signal potreben. AC15 je povezan na vhod 13 pri obeh MX.

Na osnovi zgornje realizacije KA z ROM in MX je mogoče podati naslednja opažanja. Prvič, relativno malo stanj uporablja večskočno vejitev pri določanju naslednjega stanja. V primeru preprostega procesorja je to le stanje OD (ozr. S_4). To pomeni, da bi bilo precej potratno, če bi naslavljali ROM z vsemi vhodi v KA. In drugič, vhodov je precej manj kot stanj. V našem primeru imamo le 4 vhode in 16 stanj. Ker KA opazuje le tri vhodne kombinacije



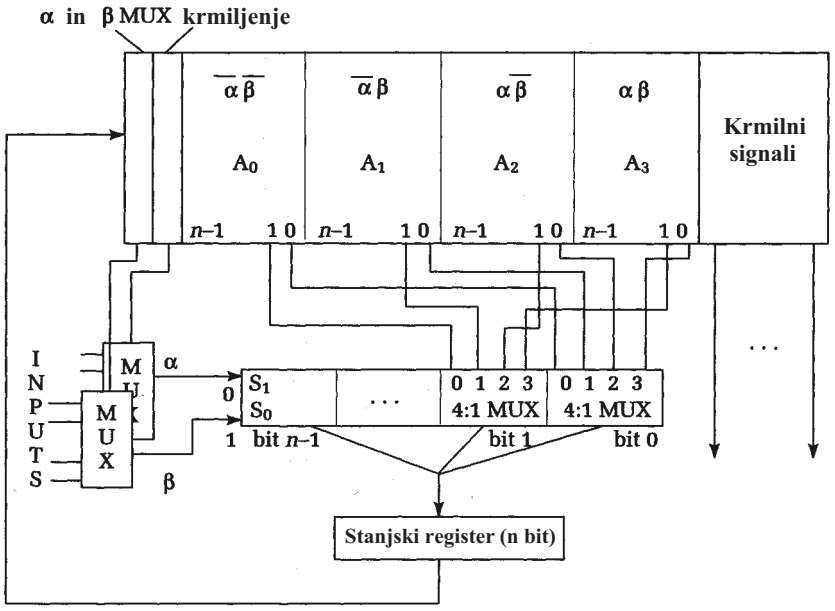
Slika 8.17: Realizacija funkcij α in β z MX.

(IR<15:14>, AC<15>, WAIT), je uporaba MX 16/1 zelo potratna. To nas navaja na naslednjo varianto skočnega sekvenčnika.

8.4.1 Horizontalni skočni sekvenčnik

Namesto ozke in dolge strukture ROM vezja (Slika 8.15), lahko za ceno dodatnih MX nadomestimo ROM s širokim in kratkim ROM elementom, kar prikazuje Slika 8.18.

Prišli smo do variante, ki se v dveh pomembnih stvareh razlikuje od prejšnje. MX so krmiljeni s kodiranimi signali iz ROM namesto iz stanj direktno. Štiri možna naslednja stanja so podana horizontalno namesto v štirih zaporednih lokacijah ROM. Če ima KA veliko število stanj toda malo vhodov, potem krmilni biti MX v ROM omogočajo uporabo takšnih MX, ki imajo manj vhodov za logiko naslednjega stanja. Ti kodirani biti so odvisni le od stanj, saj le stanje določa adreso ROM-a. Zato takšna horizontalna implementacija implicira Moorov KA.



Slika 8.18: Horizontalna organizacija ROM sekvenčnika.

Primerjava vertikalne in horizontalne organizacije ROM nas pripelje do naslednjega zaključka. Pri vertikalni varianti smo imeli MX 16/1 (2x) s 4 selektorskimi vhodi. Pri horizontalni varianti pa imamo MX 2/1 (2x) z 1 selektorskim vhodom. Dimenzija ROM je v vertikalni varianti enaka: (16 x 4) x 18 = 1152 bitov, v horizontalni varianti pa: 16 x 32 = 16 x (14 + 4 x 4 + 2) bitov = 512 bitov.

8.5 Mikroprogramiranje

Mikroprogramiranje je poseben pristop k implementaciji krmiljenja v okviru procesne enote (CPU), kjer so tako stanja v naslednjem trenutku kot tudi izhodni signali shranjeni v ROM.

Pri tem obstajata dve variaciji mikroprogramiranja, ki ju imenujemo horizontalno in vertikalno mikroprogramiranje. V prejšnjem poglavju smo že spoznali nekaj distinkcij med horizontalno in vertikalno organizacijo logike za določitev naslednjega stanja. V tem poglavju bomo podali razliko še med horizontalno in vertikalno realizacijo izhodne logike, ki določa izhodne signale krmilne enote.

Pri horizontalnem mikroprogramiranju obstaja za vsak izhodni signal en ROM izhod, kar pomeni en stolpec v pripadajoči pomnilni tabeli, ki jo hrani ROM. Vertikalno mikroprogramiranje pa upošteva dejstvo, da vsi izhodni signali niso aktivni hkrati v istem stanju. Zato so izhodni krmilni signali zakodirani, kar reducira širino ROM besede (vrstice v ustrezni tabeli), na račun dodatne logike za dekodiranje.

8.5.1 Horizontalno mikroprogramiranje

Horizontalna organizacija izračuna naslednjega stanja, ki smo jo spoznali na Sliki 8.18, predstavlja jedro horizontalnega mikroprogramskega krmilnika.

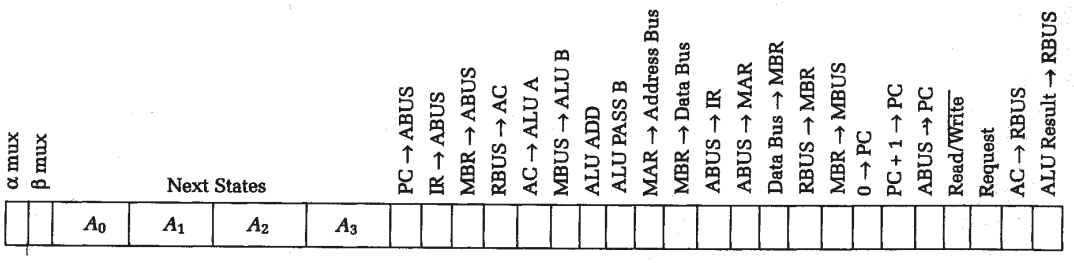
V primeru preprostega testnega procesorja smo imeli 14 registrskih prenosov, ki smo jih preslikali v 22 diskretnih mikrooperacij.

Mikrooperacije preprostega procesorja:

PC → ABUS
IR → ABUS
MBR → ABUS
RBUS → AC
AC → ALU A
MBUS → ALU B
ALU ADD
ALU PASS B (prepusti B vhod na ALU izhod)
MAR → Address Bus
MBR → Data Bus
ABUS → IR
ABUS → MAR
Data Bus → MBR
RBUS → MBR
MBR → MBUS
0 → PC
PC + 1 → PC
ABUS → PC
READ/$\overline{\text{WRITE}}$
REQ
AC → RBUS

ALU R → RBUS

Širina besede ROM je v primeru horizontalnega mikroprogramiranja enaka 2 + 4 x 4 + 22 = 40 bitov, kar prikazuje tudi Slika 8.19.



Slika 8.19: Beseda horizontalne mikrokode v primeru preprostega procesorja.

Pri tem so podatkovni vhodi α MX enaki WAIT (Sel_0) in IR<15> (Sel_1), β MX pa AC<15> (Sel_0) in IR<14> (Sel_1). Resetiranje stanjskega registra je izvedeno direktno. Kompletno ROM vsebino za Moorov mikroprogramski krmilnik podaja Slika 8.20.

Trenutno stanje (naslov)	α mux β mux	Naslednja stanja A_0	A_1	A_2	A_3	PC → ABUS	IR → ABUS	MBR → ABUS	RBUS → AC	AC → ALU A	MBUS → ALU B	ALU ADD	ALU PASS B	MAR → Address Bu	MBR → Data Bus	ABUS → IR	ABUS → MAR	Data Bus → MBR	RBUS → MBR	MBR → MBUS	0 → PC	PC + 1 → PC	ABUS → PC	Read/Write	Request	AC → RBUS	ALU Result → RBUS
RES (0000)	0 0	0001	0001	0001	0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
IF_0 (0001)	0 0	0010	0010	0010	0010	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
IF_1 (0010)	0 0	0010	0010	0011	0011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
IF_2 (0011)	0 0	0100	0100	0011	0011	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0
IF_3 (0100)	0 0	0100	0100	0101	0101	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
OD (0101)	1 1	0110	1001	1011	1110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LD_0 (0110)	0 0	0111	0111	0111	0111	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
LD_1 (0111)	0 0	1000	1000	0111	0111	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0
LD_2 (1000)	0 0	0001	0001	0001	0001	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1
ST_0 (1001)	0 0	1010	1010	1010	1010	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0
ST_1 (1010)	0 0	0001	0001	1010	1010	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0
AD_0 (1011)	0 0	1100	1100	1100	1100	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
AD_1 (1100)	0 0	1101	1101	1100	1100	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0
AD_2 (1101)	0 0	0001	0001	0001	0001	0	0	0	1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
BR_0 (1110)	0 1	0001	1111	0001	1111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BR_1 (1111)	0 0	0001	0001	0001	0001	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Slika 8.20: Vsebina ROM za Moorov mikroprogramski krmilnik.

Horizontalno mikroprogramiranje ponuja maksimalno fleksibilnost, saj v vsakem trenutku (stanju) omogoča dostop do vseh krmilnih točk. Slabost pa je v precejšnji širini besede, ki gre lahko v nekaj sto bitov pri kompleksnih krmilnikih.

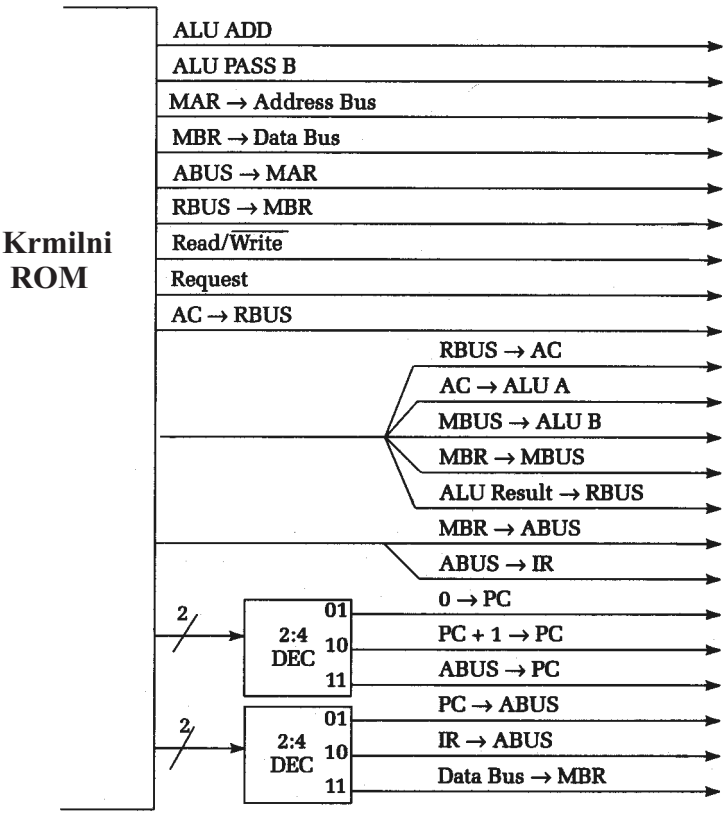
Redukcijo širine besede lahko uspešno izvedemo s kodiranjem izhodov, ki niso aktivni hkrati. N.pr. $0 \rightarrow$ PC in PC $+ 1 \rightarrow$ PC sta že takšen primer akcij, ki se ne moreta izvesti hkrati. Tudi PC $\rightarrow$ ABUS in IR $\rightarrow$ ABUS sta takšen primer. Kot primer kodiranja izhodnih krmilnih signalov vzemimo tri mikrooperacije nad PC registrom: $0 \rightarrow$ PC, PC $+ 1 \rightarrow$ PC in ABUS $\rightarrow$ PC. Če vzamemo še slučaj, ko se nobena od zgornjih mikrooperacij ne izvrši in ga označimo z Ni PC krmiljenja (angl. No PC control), potem lahko za štiri slučaje uporabimo kodiranje:

Koda	Pomen
00	Ni PC krmiljenja
01	$0 \rightarrow$ PC
10	PC $+ 1 \rightarrow$ PC
11	ABUS $\rightarrow$ PC

Za obravnavani preprost procesor je še kar nekaj strategij za kodiranje izhodov. N.pr. MAR $\rightarrow$ Address Bus in REQ sta vedno aktivna skupaj, kot tudi RBUS $\rightarrow$ AC, MBUS $\rightarrow$ ALU B, MBR $\rightarrow$ MBUS in ALU $\rightarrow$ RBUS. Kodirani ROM za slučaj mikroprogramskega krmilnika preprostega procesorja prikazuje Slika 8.21.

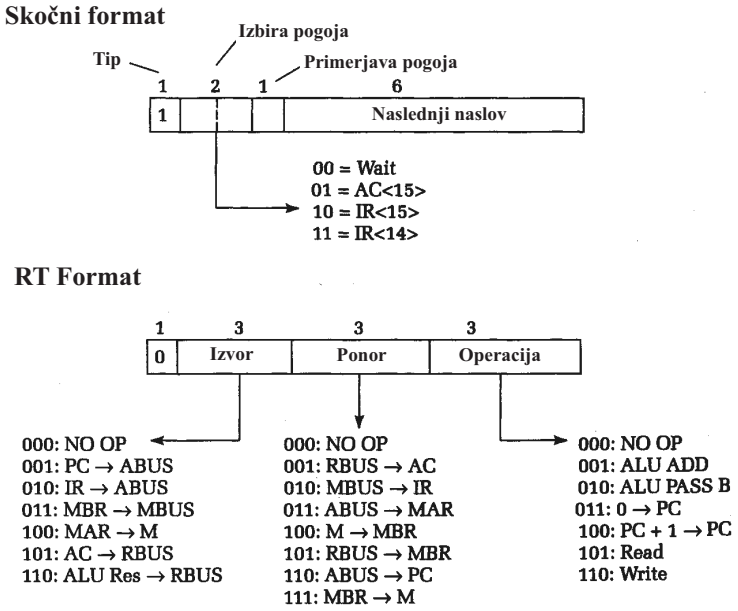
8.5.2 Vertikalno mikroprogramiranje

Za čimboljše kodiranje vsebine ROM se uporabljajo različni formati mikrobesed. To pomeni, da ima vsak format svojo strukturo podatkov. N.pr. mnogo stanj ne potrebuje pogojnega skoka v naslednje stanje, saj se enostavno stanje preslika v naslednjega v sekvenci, ki ima kodo za eno večjo od kode trenutnega stanja. To lahko rešimo tako, da vpeljemo dva formata, eden za pogojne skoke in drugi za registrske prenose. Slika 8.22 prikazuje omenjena dva formata mikroprogramske besede in pomen posameznih delov v formatih.



Slika 8.21: Kodirana vsebina ROM za mikroprogramski krmilnik.

Vsebina ROM v primeru vertikalnega mikroprogramiranja z dvema formatoma je podana na Sliki 8.23.



Slika 8.22: Dva formata pri vertikalnem mikroprogramiranju.

ROM naslov	Simbolična vsebina		Binarna vsebina
000000	RES	RT PC→ MAR, PC + 1 → PC	0 001 011 100
000001	IF0	RT MAR → M, Read	0 100 000 101
000010		BJ Wait=0, IF0	1 000 000 001
000011	IF1	RT MAR → M, M → MBR, Read	0 100 100 101
000100		BJ Wait=1, IF1	1 001 000 011
000101	IF2	RT MBR → IR	0 011 010 000
000110		BJ Wait=0, IF2	1 000 000 101
000111		RT IR → MAR	0 010 011 000
001000	OD	BJ IR<15>=1, OD1	1 101 010 101
001001		BJ IR<14>=1, ST0	1 111 010 000
001010	LD0	RT MAR → M, Read	0 100 000 101
001011	LD1	RT MAR → M, M → MBR, Read	0 100 100 101
001100		BJ Wait=1, LD1	1 001 001 011
001101	LD2	RT MBR → AC	0 110 001 010
001110		BJ Wait=0, RES	1 000 000 000
001111		BJ Wait=1, RES	1 001 000 000
010000	ST0	RT AC → MBR	0 101 101 000
010001		RT MAR → M, MBR → M, Write	0 100 111 110
010010	ST1	RT MAR → M, MBR → M, Write	0 100 111 110
010011		BJ Wait=0, RES	1 000 000 000
010100		BJ Wait=1, ST1	1 001 010 010
010101	OD1	BJ IR<14>=1, BR0	1 111 011 101
010110	AD0	RT MAR → M, Read	0 100 000 101
010111	AD1	RT MAR → M, M → MBR, Read	0 100 100 101
011000		BJ Wait=1, AD1	1 001 010 111
011001	AD2	RT AC + MBR → AC	0 110 001 001
011010		BJ Wait=0, RES	1 000 000 000
011011		BJ Wait=1, RES	1 000 000 000
011100	BR0	BJ AC <15>=0, RES	1 010 000 000
011101		RT IR → PC	0 010 110 000
011110		BJ AC<15>=1, RES	1 011 000 000

Slika 8.23: Vsebina vertikalnega mikroprograma.

Literatura

[1] Andrej Dobnikar: Logične strukture in sistemi I, Založba FE in FRI, 2009.

[2] Randy H. Katz: Contemporary Logic Design, Benjamin/cummings Publish. Comp., Inc., 1994.

[3] Yale N. Patt, Sanjay J. Patel: Introduction to Computing Systems, from bits & gates to C and beyond, McGraw Hill, Higher Education, 2005.

[4] David A. Patterson, John L. Hennessy: Computer Organization and Design, Morgan Kaufmann Publish., 2005.

[5] Sebastian Coope, John Cowley, Neil Willis: Computer Systems, Architecture, networks and communications, McGraw Hill, 2002.